

Provides a practical understanding of the core Linux system function calls and programming interfaces.

Covers key Linux technologies including processes, sockets, POSIX threads, interprocess communication, and file input and output.

Gives an exhaustive API reference to the most important Linux functions.

Included on the companion CD are Kylix Open Edition and Kylix 2 Trial Edition.



Companion
CD-ROM
Included



The Tomes of Kylix™

The Linux® API

Glenn Stephens

The Tomes of Kylix™ The Linux® API

Glenn Stephens

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Stephens, Glenn

The tomes of Kylix: the Linux API / by Glenn Thomas Stephens.

p. cm.

Includes bibliographical references and index.

ISBN 1-55622-823-6 (pbk.)

1. Application software. 2. Kylix. 3. Linux. I. Title.

QA76.76.A65 S76 2002

005.2'82--dc21

2001057353

CIP

© 2002, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard

Plano, Texas 75074

No part of this book may be reproduced in any form or by any means
without permission in writing from Wordware Publishing, Inc.

Printed in the United States of America

ISBN 1-55622-823-6

10 9 8 7 6 5 4 3 2 1

0111

Kylix is a trademark of Borland Software Corporation.

Linux is a registered trademark of Linus Torvalds.

Other product names mentioned are used for identification purposes only and may be trademarks of their respective companies.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Contents

	Acknowledgments	xiii
	Introduction	xv
	About the Author	xxi
Chapter 1	A Linux API Primer	1
	Linux History	1
	Applications — What Developers Do	2
	Introducing the Linux API	5
	What is Kylix?	6
	What Windows Has That Linux Doesn't and Vice Versa	7
Chapter 2	Linux API Errors	9
	Dealing with Linux API Errors	9
	Signals as Errors	10
	Linux API Error Functions	10
	errno	10
	__errno_location	11
	perror	11
	strerror	13
	Linux API Error Codes	14
	Errors are Not Exceptions	18
	Conclusion	26
Chapter 3	Linux Input and Output	27
	Introduction	27
	File Descriptors and Streams	28
	Input and Output Methods	29
	Opening and Closing Files	29
	Reading and Writing Files	30
	Standard Input, Standard Output, and Standard Error	33
	Converting File Descriptors to File Streams and Vice Versa	34
	Transferring Handles between Kylix and the API Reference	35
	Converting a File Descriptor into a Stream	36
	File Permissions and Ownership	38
	Getting Information and Attributes about a File	42
	Scanning Files in a Directory	45
	Working with Buffers	46
	File Operations (Renaming, Copying)	46
	The ioctl Function	47
	Creating Links (Shortcuts for Linux)	48
	File Conventions	48
	Unlocked Functions	49

varargs — A New Declaration for Kylix	49
File Errors	50
API Reference	51
__chdir	51
__close	51
__mkdir	52
__read	53
__rename	54
__truncate	55
__write	55
alphasort, alphasort64	56
chmod	57
clearerr, clearerr_unlocked	58
closedir	59
creat, creat64	60
dirfd	61
fchmod	62
fclean	63
fclose	64
fcloseall	64
fcntl	65
fdopen	67
feof, feof_unlocked	69
ferror, ferror_unlocked	70
fflush, fflush_unlocked	70
fgetc, fgetc_unlocked	71
fgetpos, fgetpos64	72
fgets, fgets_unlocked	73
fileno, fileno_unlocked	75
fopen, fopen64	76
fprintf	77
fputc, fputc_unlocked, putc	78
fputs, fputs_unlocked	79
fread, fread_unlocked	80
freopen, freopen64	82
fseek, fseeko, fseeko64	83
fsetpos, fsetpos64	84
fstat, fstat64	85
ftell, ftello, ftello64	86
fwrite, fwrite_unlocked	87
get_current_dir_name	89
getc, getc_unlocked	89
getcwd	90
getdelim	91
getline	93
getw	94
getwd	95
ioctl	96

lseek, lseek64	97
lstat, lstat64	98
open, open64	100
open_memstream	102
opendir	103
pread	103
putw	104
pwrite	105
readdir, readdir64	105
remove	107
rewind	107
rewinddir	108
scandir, scandir64	109
seekdir	111
setbuf	112
setbuffer	113
setlinebuf	114
setvbuf	115
stat, stat64	116
telldir	117
tempnam	118
tmpfile, tmpfile64	119
tmpnam, tmpnam_r	120
ungetc	120
versionsort, versionsort64	122
Practical Example — A dBase III File Reader	122
Where to Go from Here	132
Chapter 4 Processes	133
Introduction	133
What is a Process?	133
Gathering Information about a Process under Linux	134
Processes and Linux Environment Variables	136
An Environment Example	137
Forking Processes	140
Making Use of the Forked Process	142
Process Groups	143
Waiting on Other Processes	144
Terminating a Process	147
Executing Linux Commands	149
Using the exec Functions	150
Enumerating Users and Groups from within Code	153
Running Normal Apps with Permission of the App's Owner	156
Creating Daemons for Linux	157
API Reference	160
__secure_getenv	160
clearenv	161
daemon	161

endgrent	162
endpwent	162
execl	163
execle	163
execlp	165
execv	166
execve	168
execvp	170
fexecve	171
fork	172
getegid	173
getenv	173
geteuid	174
getgid	175
getgrent	175
getgrgid	177
getgrnam	178
getgroups	179
getpgid	180
getpgrp	180
getpid	181
getppid	181
getpwent	182
getpwnam	183
getpwuid	184
getuid	185
group_member	186
putenv	187
setegid	188
setenv	189
seteuid	190
setgid	191
setgrent	192
setpgid	192
setpgrp	193
setpwent	194
setregid	194
setreuid	195
setuid	196
system	197
unsetenv	198
vfork	198
wait	199
wait3	201
wait4	203
waitpid	204
A Practical Look at Processes — A Server Health Check	205
Conclusion	210

Chapter 5	Interprocess Communication (IPC)	211
	Introduction	211
	The Need to Communicate between Processes	211
	What Methods are Available to Us?	212
	An In-depth Look at the Communication Methods	215
	Signals — What are They?	215
	Catching a Signal	216
	Sending a Signal to Another Process	217
	Using Signal Sets — The “Reliable” Method	218
	Blocking Signals	221
	Using Signals within Kylix Applications	223
	Programming Pipes	225
	Named Pipes — A Better Alternative	229
	Message Queues — A Better Way	233
	Semaphores — Communicating Operation Status Between Processes	240
	Programming Semaphores	240
	Shared Memory	245
	Programming Shared Memory	246
	Using Semaphores with Shared Memory	248
	Creating Unique IPC Keys	253
	Which Methods Work Best for Different Kinds of Applications	254
	API Reference	255
	raise	255
	alarm	255
	kill	256
	killpg	257
	mkfifo	258
	msgctl	259
	msgget	260
	msgrcv	261
	msgsnd	262
	pause	263
	pclose	263
	pipe	264
	popen	265
	psignal	266
	semctl	267
	semget	271
	semop	271
	shmat	273
	shmctl	274
	shmdt	276
	shmget	276
	sigaction	277
	sigaddset	278
	sigandset	279
	sigdelset	280
	sigemptyset	280

sigfillset	281
siggetmask	282
sighold.	282
sigignore	283
sigisemptyset	283
sigismember.	284
signal	285
sigorset	286
__sigpause.	287
sigpending.	288
sigprocmask.	288
sigrelse	290
sigset	291
sigsuspend.	292
sigwait.	293
strsignal	294
sigtimedwait.	294
A Practical Use — A Message Queue Component	295
Conclusion	303

Chapter 6 POSIX Threads 305

Introduction	305
What are Threads? — An Introduction.	305
LinuxThreads — The POSIX Standard on Linux.	306
Understanding Threads within a Kylix Application	306
Choosing between Threads and Processes	308
Error Handling with the POSIX Thread Functions	309
Reentrant Functions.	309
Creating Threads	310
Thread Attributes	311
Waiting for Threads to Finish.	317
Killing Threads	318
Synchronization Methods Using POSIX Threads	319
Semaphores	326
Condition Variables	333
Listen Carefully, I Shall Say This Only pthread_once	335
Other Synchronization Methods.	336
Thread-Specific Data	348
Signals and Threads.	351
Integrating with CLX's TThread Object	352
API Reference.	354
__pthread_initialize.	354
__pthread_cleanup_pop	354
__pthread_cleanup_pop_restore	355
__pthread_cleanup_push.	355
__pthread_cleanup_push_defer	356
clone.	357
GetCurrentThreadID	358

pthread_atfork.	358
pthread_attr_destroy	360
pthread_attr_getdetachstate	361
pthread_attr_getguardsize	362
pthread_attr_getinheritsched	362
pthread_attr_getschedparam	363
pthread_attr_getschedpolicy	364
pthread_attr_getscope	365
pthread_attr_getstack	365
pthread_attr_getstackaddr	366
pthread_attr_getstacksize	367
pthread_attr_init	367
pthread_attr_setdetachstate	368
pthread_attr_setguardsize	368
pthread_attr_setinheritsched	369
pthread_attr_setschedparam	370
pthread_attr_setschedpolicy	371
pthread_attr_setscope	372
pthread_attr_setstack	372
pthread_attr_setstackaddr	373
pthread_attr_setstacksize	374
pthread_barrier_destroy	374
pthread_barrier_init	375
pthread_barrier_wait	376
pthread_barrierattr_destroy	376
pthread_barrierattr_getpshared	377
pthread_barrierattr_init	377
pthread_barrierattr_setpshared	378
pthread_cancel	378
pthread_cond_broadcast	380
pthread_cond_destroy	380
pthread_cond_init	381
pthread_cond_signal	382
pthread_cond_timedwait	382
pthread_cond_wait	383
pthread_condattr_destroy	384
pthread_condattr_getpshared	384
pthread_condattr_init	385
pthread_condattr_setpshared	385
pthread_create	386
pthread_detach	387
pthread_equal	389
pthread_exit	390
pthread_getschedparam	391
pthread_getspecific	392
pthread_join	393
pthread_key_create	393
pthread_key_delete	394

pthread_kill	395
pthread_kill_other_threads_np	396
pthread_mutex_destroy	396
pthread_mutex_init	396
pthread_mutex_lock	397
pthread_mutex_timedlock	398
pthread_mutex_trylock	399
pthread_mutex_unlock	399
pthread_mutexattr_destroy	400
pthread_mutexattr_getpshared	400
pthread_mutexattr_gettype	401
pthread_mutexattr_init	402
pthread_mutexattr_setpshared	402
pthread_mutexattr_settype	403
pthread_once	404
pthread_rwlock_destroy	404
pthread_rwlock_init	405
pthread_rwlock_rdlock	406
pthread_rwlock_timedrdlock	407
pthread_rwlock_timedwrlock	407
pthread_rwlock_tryrdlock	408
pthread_rwlock_trywrlock	409
pthread_rwlock_unlock	409
pthread_rwlock_wrlock	410
pthread_rwlockattr_destroy	410
pthread_rwlockattr_getpshared	411
pthread_rwlockattr_init	412
pthread_rwlockattr_setpshared	412
pthread_self	413
pthread_setcancelstate	413
pthread_setcanceltype	417
pthread_setschedparam	417
pthread_setspecific	419
pthread_sigmask	419
pthread_spin_destroy	420
pthread_spin_init	421
pthread_spin_lock	421
pthread_spin_trylock	422
pthread_spin_unlock	422
pthread_testcancel	423
pthread_yield	423
sem_close	424
sem_destroy	424
sem_getvalue	425
sem_init	425
sem_open	426
sem_post	427
sem_timedwait	428

sem_trywait	428
sem_unlink	429
sem_wait	429
Conclusion	430
Chapter 7 The Socket API Functions	431
Introduction	431
What is a Socket?	431
Steps in Establishing a Socket Connection.	432
Creating a Socket	433
Socket Addresses	434
Connecting a Socket Client to a Server	436
Having a Conversation with a Socket Server	439
Creating a Socket Server	442
Creating a Socket and Listening Address for a Server Socket	443
Giving the Socket a Name.	445
Listening to and Accepting Connections	446
Ways to Process a Client Socket	448
Look Up in the Sky, It's the Internet Socket Server	453
Integrating with the Visual Socket Components	455
API Reference.	456
accept	456
bind	456
connect	458
endhostent.	459
endservent.	459
gethostbyaddr	459
gethostbyname	461
gethostbyname2.	463
gethostent	464
gethostname	465
getpeername.	466
getprotobyname.	467
getprotobynumber.	468
getservbyname	469
getservbyport	470
getservent	472
getsockname	473
getsockopt.	475
htonl.	477
htons.	478
inet_addr	479
inet_ntoa	480
listen.	480
ntohl.	481
ntohs.	481
recv	482
recvfrom.	483

select	484
sethostent	490
setservent	490
send	491
sendto	492
setsockopt	495
shutdown	496
socket	497
socketpair	498
Where to Go From Here	501
Chapter 8 Importing Other APIs	503
Introduction	503
Getting Starting with Shared Objects.	503
Creating a Shared Object with Kylix.	504
Using the Shared Object Created with Kylix	506
Creating a Shared Object in C	508
Rules for Importing Functions from a C-Written Library	509
Using a C-Created Shared Object in Your Application.	511
Dynamically Loading a Shared Object	512
Shared Object API Functions.	517
dladdr	517
dlclose.	517
dLError.	518
dlopen	519
dlsym	520
Conclusion	520
Appendix References for Kylix Development.	521
Web Sites	521
Books	523
Index.	525

Acknowledgments

I would like to start by thanking the Wordware team who helped me through the long process of writing a reference book for the Kylix developer. The main people I would like to thank are Jim Hill and Wes Beckwith. To Jim, thank you for making my dream of writing a book a reality and for your patience and assistance through the entire process. To Wes, thanks for putting up with the excessive questions from the author on the other side of the planet and for your helpful nature.

I would also like to thank my technical reviewers, Malcolm Groves and Chad (Kudzu) Hower, for their fantastic effort in reviewing the book. You really helped a lot, and the book would not be the same without your input and feedback. Malcolm, I would like to thank you for your help, even though you had a million other things to do. To Chad, thank you for reviewing the remaining chapters at the eleventh hour and fitting in the reviewing in your busy schedule whenever you had a chance. Surviving a technical review about a chapter on sockets from the writer of WinShoes and Indy is an honor in itself.

I would also like to thank my family, friends, and associates for their support and encouragement throughout this book's development. Special thanks also go to Pierre and Kirsty, Dion and Libby, Mark and Dr. Clare, the staff at Borland Australia, and the Australian Delphi User Group.

Last but not least, I would like to thank Rebel and Zai for your patience, love, and understanding while I went off the beaten track to write this book. For this reason I dedicate this book to the both of you.

Introduction

Linux is a free operating system based on UNIX. Linux started out as a hobby project of Linus Torvalds when he was convinced that the MINIX operating system that his university taught could be improved upon. With the assistance of developers worldwide on the Internet, Linus was able to develop one of the most powerful and robust operating systems in the last few years.

Now Linux is a powerful operating system that competes head to head with many of the top server operating systems such as Windows NT and other versions of UNIX. Because of its low cost, scalability, and network readiness, Linux has carved for itself a powerful position in the server market, especially with database, e-commerce, and Web servers.

Not only is Linux a powerful server operating system; it is also a fantastic desktop environment with a powerful user interface, similar to the styles of the Macintosh or Windows. Today, there are many powerful software products available for Linux that have all the functionality of their non-Linux counterparts such as word processors, accounting applications, databases, and much more.

Although there are thousands of applications available today for Linux, this still does not compare to the amount of software available for Windows, but this is changing. Much of Linux's grandeur comes from its ability to develop server applications. Now with Kylix, for the first time, developers for Linux will have a powerful rapid application development tool for generating client-side applications with the same speed as their Windows friends. Borland often bragged publicly that the day that Kylix came out, the number of Linux desktop applications available would almost double. While Kylix can develop great desktop applications for Linux, Kylix also makes it easy to create industrial strength server applications. Behind Kylix's strength is the fact that it can connect directly to the core API functions of Linux to fully push a Linux application to its limits.

Enter the Linux API. The Linux API is an extremely powerful set of functions and is based on a solid evolution of software development standards over the last several decades. Linux itself is based on UNIX, which was developed at Bell Labs by Ken Thompson. Bell also developed the C language and developed the UNIX operating system completely in C. As a result, the core API consists of some of the standard C library and the features of the UNIX system such as process management, UNIX-based file handling, sockets, POSIX threads, and more.

Originally the library that consisted of the Linux API was compiled into the application, much like Kylix will compile a Pascal unit into your application. Later, as Linux grew, it began to implement many of the same industry standards that UNIX uses, which include the POSIX, IEEE, and X/Open standards.

The POSIX library is very heavily used within the Linux API. POSIX itself covers the areas that a function library needs to implement in order to make professional, mission-critical applications. Process management, threading, and interprocess communication are just some of the areas that the POSIX library implements.

Once the Linux API calls were mostly defined, the libraries that consisted of the Linux API moved from being statically linked to being a shared object library that any application could link to. It still consisted of the main functions but now had the ability to be used by any application, not just C and C++ applications. This was one of the main jumps that allowed Kylix to use the Linux API functions. As the functions of Linux can be used from Kylix, we have the full power of the Linux API.

From this convergence of UNIX calls, POSIX, and X/Open calls, the library was placed under an open source license and is officially called the GNU Lib C. GNU (an ironic acronym for “GNU’s not UNIX”) is an open source community of developers all creating software, tools, and documentation to benefit the software community.

You will often find that the Linux API will have the names “GNU LibC,” “LibC,” or “Linux API.” All of these names can be interchanged.

History of Kylix Development

Everyone has been talking about Kylix. Kylix is Borland’s reincarnation of its popular Delphi tool for the Linux platform. Delphi is a powerful rapid application development (RAD) environment for the Windows operating system. Delphi’s power comes from the fact that it has built within it a reusable and powerful object model known as the Visual Component Library (VCL), direct access to the Windows API, powerful database support, and a way to build applications faster than any method I have seen.

Kylix loses none of the benefits of Delphi. It still has the same strong object model based on the VCL, but now implements CLX (cross-platform library system), which is a cross-platform version of the VCL for both Windows and Linux. It still has the same strong database support but now connects through to Linux databases such as MySQL and InterBase, among others. Just as Delphi has strong links into the Windows API, Kylix has the ability to access the Linux API in the same manner. It is the fact that Kylix can access the Linux API that makes it such a powerful tool.

There are drawbacks to coding with the Linux API, however. The first and most apparent drawback is that you lose the cross-platform nature that CLX provides. You do not have directly portable code that you can move between Delphi and Kylix. Coding with the Linux API means you also lose the ease of use that you may be familiar with using VCL or CLX.

But there are benefits, too — many, many benefits. By using the Linux API, you have access to the powerful features of Linux. You can take advantage of managing the specific aspects of Linux by being able to use the built-in functions that the Linux API provides. You can analyze the system or provide features that the CLX libraries do not.

The Linux API is a powerhouse of functions that allows a developer to manage processes, files, threads, sockets, system management, and much more. The Linux API, which is also known as GLIBC or simply LibC, is a collection of functions that implement the POSIX standard set out by IEEE. POSIX is designed as a platform-independent

programming interface so that developers can code an application and theoretically recompile the application on another platform to make it run.

Why Should I Work with the Linux API?

Someone once said, “Knowledge is power.” When it comes to software development, the knowledge that is gained learning the underlying system API gives you the power to design and develop more efficient applications for yourself and your customers. By looking at how the system can work as a whole, you will be in a better position to create solutions for your customers and a better understanding will also assist you in analyzing any bugs that may be found in your new Linux apps.

Chapter Summaries

Chapter 1 — A Linux API Primer defines what the Linux API is as well as provides a brief discussion on the evolution of the Linux API. This chapter also discusses how Kylix can work with the Linux API.

Chapter 2 — Linux API Errors lists the error codes that can be returned by most of the Linux API calls. This chapter also demonstrates techniques that you can use to merge Linux API error messages with Kylix’s normal exception model.

Chapter 3 — Linux Input and Output examines the Linux API calls for working with basic input and output and then working with files and directories. Beginning with reading and writing files and streams, this chapter examines methods for navigating directories, file locking, and obtaining optimum performance.

Chapter 4 — Processes looks at creating, calling, and manipulating Linux processes. The concepts of processes, process groups, and forking are introduced along with examining the Linux user environment.

Chapter 5 — Interprocess Communication (IPC) deals with methods of communicating between different processes such as signals, pipes, named pipes, message queues, and shared memory. The benefits and pitfalls of each method of communication are examined along with ways of integrating these methods into your normal Kylix CLX calls.

Chapter 6 — POSIX Threads covers the methods of multitasking under Linux within the same process. This chapter examines when it is appropriate to use threads and processes, synchronization methods between threads, and integrating thread calls with the CLX library.

Chapter 7 — The Socket API Functions studies direct TCP/IP communication with Linux and the methods for creating fast and efficient socket clients and servers and discusses methods of integrating direct API calls with Kylix’s TClientSocket and TServerSocket, and the new NetCLX components.

Chapter 8 — Importing Other APIs gives you the knowledge to transform many of Linux’s existing APIs written in C to Object Pascal that can be called from your application. This is

especially useful when you need to call the vast collection of library functions not converted in Kylix.

Appendix — References for Kylix Development lists useful books and URLs to get solid information on Linux and Kylix development.

Conventions

The function calls are laid out to provide as much information as possible to the reader. Each function is specified with each parameter on a separate line. A listing of the function call, parameters, and any return values are described in detail along with an example. Readers of other Tomes of Delphi books will already find this format quite familiar. The sample below demonstrates the format of a typical function and shows the function name, the unit where the function is located, the function's syntax, a description outlining the use of the function, a list of the parameters, the return values, a list of related functions, and, in most cases, an example outlining the use of the function. In this book all the Linux API functions documented are declared in the `LibC.pas` unit.

getenv ***LibC.pas***

Syntax

```
function getenv(  
  Name: Pchar  
): PChar;
```

Description

The `getenv` function returns the value of an environment name/value pair. A typical environment variable is of the form "HOME=/root." A list of environment variables can typically be viewed by using the `env` command.

Parameters

Name: Name refers to the name of the environment value in the name/value pair.

Return Value

The function returns the value of the environment name/value pair.

See Also

`setenv`, `putenv`, `clearenv`, `__secure_getenv`

Example

```
var  
  MailDir: string;  
  
begin  
  //Find out the location of the Mail directory.  
  MailDir := StrPas(getenv('MAIL'));
```

```
    showmessage('Mail for the user can be found in ' + MailDir);
end;
```

Many chapters also include demonstration applications towards the end of each chapter. These are practical demonstrations that give you reusable code that you can apply to your Kylix applications.

All of these demonstration applications use the standard Delphi syntax in a fixed-width font similar to the listing below.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    counter: integer;
begin
    for counter := 1 to 10 do
    begin
        //Add an item
        ListBox1.Items.Add(inttostr(counter));
    end;
end;
```

What This Book Covers

The goal of this book is to give you a practical understanding of the core Linux system function calls and the programming interfaces by giving an in-depth look at how the Linux API functions, practical examples of the Linux API, mixing the Linux API and the Kylix CLX, and a detailed reference for the key API calls exposed by Kylix.

This book is not intended to be a Kylix guide, CLX reference, or a replacement for the Kylix manuals. This book assumes that you can program in Kylix or Delphi and that you are relatively new to Linux. Should you need other reference material, examine Wordware's extensive collection of Delphi and Kylix titles at <http://www.wordware.com>.

The Linux API also exhaustively covers areas that even Kylix has not implemented yet, although I am sure Borland will add more functions to their version of the Linux API unit calls in later releases. As a result this book only documents the major parts of the Linux API that are implemented in Kylix that you are most likely to use.

This book assumes that you are an intermediate to advanced Delphi programmer who is looking for a strong reference for the Linux API. Like any reference book, there is no need to read it from start to finish, but instead you can go directly to any chapter without the need to read any previous chapter. Reading the book from the start, however, will give you a solid understanding of the Linux API functions.

Linux is a powerful operating system, and I wish you well upon your journey to understanding the internals and architecture of the Linux API.

About the Author

Glenn Stephens is a Borland Certified Consultant who has been working with Delphi and various Pascal variants for well over 12 years. Working in application and e-commerce development, Glenn designs, develops, and deploys applications for businesses, government departments, and corporations throughout Australia.

Glenn is also a regular speaker at Borland conferences, symposiums, and user group meetings around Australia. In addition, Glenn teaches Delphi courses for Borland Australia and has written for several periodicals including *Delphi Developer* and *Australian Corporate IT* magazines.

A Linux API Primer

Welcome to the wonderful world of Linux. Those of you who have been playing with Linux for a while know that it is a powerful operating system worthy of the praises that many give it around the world every day. For those who are taking your first steps into Linux, you are in for a fun ride.

In this chapter I will take you through the various aspects of the Linux operating system, so that you will be aware of how and where you can use the Linux API functions to your benefit. You will learn about console, X Windows, and Qt applications and how they work within the Linux system. In addition to the different kinds of applications, we will look at the history of Linux as well as some practical aspects of the operating system.

Linux History

Linux had its humble beginnings when a young university man named Linus Torvalds posted the following message to the comp.os.minux newsgroup back in October 1991:

“I am doing a (free) operating system (just a hobby, won’t be big and professional like gnu) for 386(486) AT clones...”

This statement will go down in history as one of the biggest IT understatements ever. At the time Linus posted this message, Linux was nothing more than an operating system kernel with a C compiler, a bash shell, and some basic applications. From Linus’ post, programmers from around the world began assisting with the project, adding features and stability to the operating system. In addition to the features developers were putting into the operating system, many other developers were creating applications specifically to run under Linux.

Although programmers with vast knowledge of operating system principles did much of the initial work on Linux, such as kernel and driver development, everyday developers were at a point where applications could be built for the operating system similar to many of the major applications such as Office application suites, Web servers, and many others.

Today, Linux is a powerful operating system that competes head to head with many of the top server operating systems such as Windows NT and UNIX. Because of its low cost, scalability, and network readiness, Linux has carved for itself a powerful position in the server and especially the e-commerce and Web server market.

Not only is Linux a powerful server operating system, it is also a fantastic desktop environment with a rich user interface, comparable to Windows or the Macintosh. There are many powerful software products available today that have all the functionality of

non-Linux applications such as word processors, accounting applications, and high-quality databases.

Even though there are thousands of Linux applications available today, this still does not compare to the amount of software available on other platforms, such as Microsoft Windows. Much of Linux's grandeur comes from its reputation for fast and stable server applications. Kylix will, for the first time, give Linux a powerful rapid application development (RAD) tool for generating client-side applications by creating native Linux applications. As a result, Kylix can connect directly to the core Linux API in Linux's native tongue.

Applications — What Developers Do

The goal of the developer is to make applications, and there are several types of applications that can be built with Kylix. Linux normally has several kinds of applications that can be built. These are console applications, X Windows applications, daemons, and device drivers.

*Figure 1.1 -
A console
application*



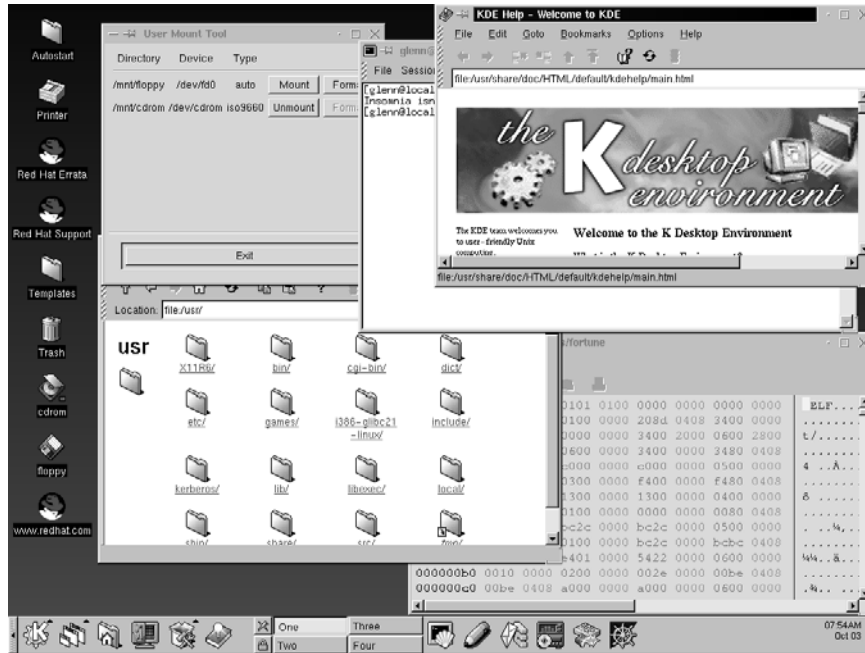
Console applications like the one shown in Figure 1.1 are text-based applications that run without the benefit of a graphical user interface. Console applications allow text-based reading and writing of information. The development of console applications today is normally only done when developing server applications that do not require a user interface or on Linux systems that have limited resources such as embedded systems. Console applications are often legacy applications from the days when Linux did not have a front-end GUI. Many Linux users who grew with the operating system are often happier using the console-based applications than their GUI counterparts.

On Linux, the graphical user interface standard is X Windows. X Windows is a user interface system that is similar to other operating system user interfaces such as Windows or Macintosh. The original X Windows display just supported basic functionality such as widgets and window control.

The user interface of X Windows has however become outdated and feels different from many of the major operating systems such as Windows or Macintosh. The next evolution for Linux on X Windows was to extend the operating system with enhanced operating system features such as drag-and-drop, a more visually pleasing interface for the user, and a

suite of applications that support the desktop system and replace many of the command-line utilities that are used with Linux. The enhancements came from groups such as KDE and GNOME. The KDE desktop manager (shown in Figure 1.2) and the GNOME desktop manager (shown in Figure 1.3) are both built on top of X Windows.

Figure 1.2 -
The KDE
desktop
manager



The KDE desktop manager is important to Kylix developers because KDE uses a set of library functions from Trolltech known as Qt. The Qt library is important because the GUI-based applications created with Kylix rely on the Qt libraries to be on the Linux system in order for applications created with Kylix to work. This is due to the fact that the KDE system was actually developed with the Qt libraries.

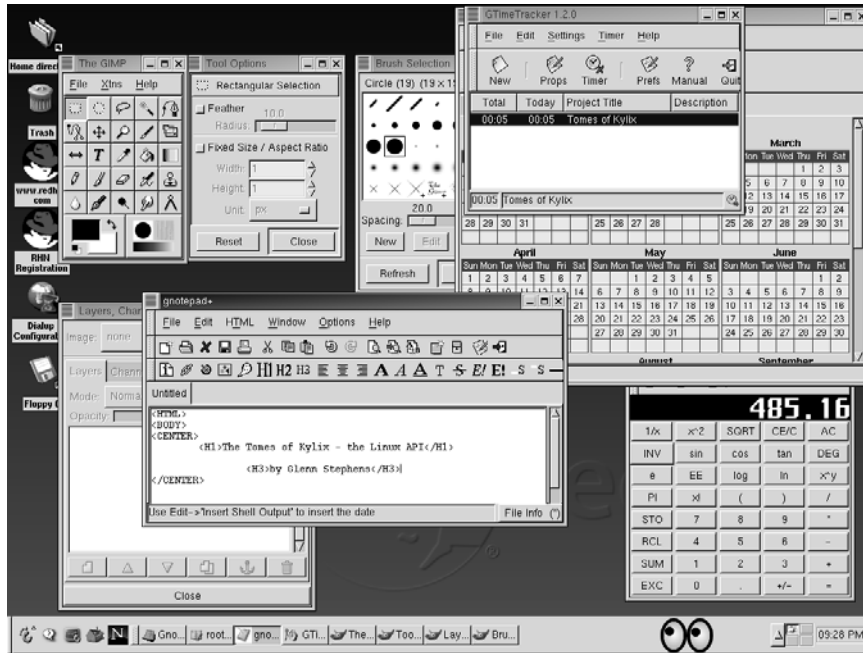
However, the Qt libraries that were used when KDE was initially developed were not open source. Open source, which involves giving away the source code to an application, is close to the heart of many Linux developers, as Linux itself was founded on an open source philosophy. As a result, many in the Linux community felt that KDE did not embody the true essence of the Linux spirit and an alternative desktop manager system for X Windows known as GNOME was created.

GNOME is similar to KDE in most aspects in that it is a friendly version of X Windows with bells and whistles similar to KDE, such as an easier to use interface, a suite of applications for the user, and graphical alternatives for the Linux command-line tools.

Although GNOME does require the use of Qt libraries to run applications, the Qt libraries can be installed on a GNOME system, and applications created with Qt for the KDE can run under GNOME. Kylix makes effective use of this fact by creating applications that use the Qt libraries so that the applications created can run on both KDE and GNOME.

Even though the applications are created with the Qt libraries, this does not mean that applications created with Kylix cannot run under simple X Windows. If Linux is running

Figure 1.3 -
The GNOME
desktop
manager



X Windows without the Qt libraries, applications can still be developed. However, the application cannot use the standard Kylix classes such as TForm and TButton. Instead, calls must be made directly to the X Windows libraries, which involves a great deal more work to get an application up and running. Just creating a simple form involves making numerous calls to the X Windows libraries. X Windows applications that do not make use of the Qt libraries are at a disadvantage, however, due to the fact that much more code needs to be written to create the application.

Daemons, pronounced “demons,” are applications that sit in the background of the operating system that usually perform some service to the Linux operating system such as providing e-mail or database services, examining the health of the Linux system, or doing other tasks that sit in the background. These applications are simple console applications that do not have a way of communicating with the visual display. Daemon applications will sit quietly in the background and process information when requested. Because the applications are console applications, they can be created with Kylix without too much trouble.

The last application type for Linux is the device driver. These are system libraries that are actually part of the Linux kernel and are used to communicate with specific devices such as hard drives, scanners, and other devices. Kylix itself is not designed to create device drivers for Linux; instead, Kylix is mainly used as a tool for creating powerful client and server applications for Linux, such as database client applications and Internet server applications.

Now that you know what kind of applications you can build with Kylix, let’s look at the library of functions that will allow you to extend these applications to their full potential — the Linux API.

Introducing the Linux API

The Linux API is an extremely powerful set of functions for Linux developers and is based on a solid evolution of software development from the last two decades. Linux itself is based on UNIX, which was developed at Bell Labs by Ken Thompson. Just before UNIX was developed, Dennis Ritchie, also from Bell Labs, developed the C language. Due to Bell's investment and commitment to UNIX, Bell was actively promoting the C language as UNIX itself was developed in C. This is much in the same way that Sun is actively promoting Java.

As a result, the core API of Linux consists of the standard C library functions, the features of the UNIX system, other development standards such as POSIX, UNIX-based file handling, sockets, and much more. Later, as Linux grew, it began to implement many of the same industry standards that UNIX implements, including the POSIX standard by IEEE and the X/Open standard.

Originally, the library that consisted of the Linux API was statically linked to a developer's application, giving the developer the benefit of having a single application. The POSIX library functions are integrated very heavily within the Linux API and consist of a solid majority of the Linux API library of functions. POSIX covers the areas that an operating system needs to implement in order to make professional, mission-critical applications.

The POSIX standards include a large number of function calls that perform a variety of standard tasks for an operating system. As POSIX is widely received in the Linux community, you will see that a large number of POSIX standards are implemented in the Linux API.

There are many different specifications involved in POSIX and the POSIX projects that incorporate functionality such as signals, process management, threading, and interprocess communication.

Once the Linux API calls were mostly defined, the libraries that consisted of the Linux API moved from being statically linked to being a shared object library that any application could link to. It still consisted of the main functions but now also had the ability to be used by any application, not just the applications written in C or C++. This was one of the main leaps that gave Kylix the ability to reuse the code of the C library. As the Linux API functions can be used from Kylix we have the same level of power at our disposal as any C or C++ developer.

From this convergence of UNIX calls, POSIX, and X/Open calls, the library was placed under an open source license and is officially called the GNU Lib C.

GNU itself is an acronym for "GNU's not UNIX," and GNU is an open source community of developers creating software, tools, and documentation to the benefit of the software community. More information on GNU can be found at <http://www.gnu.org>.

As GNU developed the official Linux API, which is based on the C libraries, you will often hear the Linux API referred to as the "GNU C Library." Other names for the library include GLibC, LibC, and the Linux API, and throughout this book, all of these names will mean the same thing.

What is Kylix?

Since Borland announced Kylix, the Linux community has been talking about it. Kylix is Borland's reincarnation of its popular rapid application development (RAD) Windows development tool, Delphi, for the Linux platform. Delphi has a vast amount of features to build powerful and reliable client/server and Internet applications such as a fantastic user interface for designing applications, hundreds of components that can be easily integrated into the application, a powerful reusable object model known as the Visual Component Library (VCL), direct access to the Windows API, powerful client server database support, and a way to build applications faster than any method I have seen.

By moving to Linux, Kylix loses none of Delphi's benefits. It still has the same strong object model based on the VCL, but now implements CLX which is a cross-platform equivalent of the VCL that runs on both Windows and Linux. The CLX looks and operates much like the VCL but any Windows-specific features have been removed from the VCL.

One of the coolest features of Delphi for Windows is that applications built in Delphi had the ability to link directly to the Windows API function calls. This meant that any Windows API function call could be done by making a call to a Windows system library from within Delphi. Just as Delphi can link into the Windows system libraries, Kylix has the ability to access the Linux API in the same manner. It is this ability that gives it the power to extend Kylix-made apps by using the library calls.

There are drawbacks to coding with the Linux API, however. The first and most apparent drawback is that the benefits of cross-platform development using the CLX libraries are lost. As a result, you do not have directly portable code that you can move between Delphi and Kylix. Coding with the Linux API means you also lose the elegant object model of CLX. But there are benefits, too — many, many benefits.

By using the Linux API, you have access to the inner features of Linux. You can take advantage of managing the Linux file security and be able to use the built-in encryption functions that the Linux API provides. You can analyze the system, or provide features that the CLX libraries do not. When developing server-side applications many developers attempt to push the limits of their server application in order to obtain the best performance they can possibly get, which can offset any loss in cross-platform maintainability.

Also, even though you directly lose the benefit of cross-compatible code on Linux by using the API calls, you can manage your code using compiler directives to ensure that Kylix compiles with one set of code and Delphi compiles with another.

Take, for example, the case where you would want to have a function that returns the name of a temporary file that you would use. You could code your function using the `{IFDEF LINUX}` directive to write code that will compile on both Kylix and Delphi.

Listing 1.1

```
{IFDEF LINUX}
function GetTempFileName: string;
var
    strFileName: PChar;
begin
    strFilename := tempnam(nil, 'tmp');
    Result := StrPas(strFilename);
end;
```

```

{$ENDIF}

{$IFDEF WIN32}
function GetTempFile: string;
var
    strPath, strFilename: array[0..MAX_PATH] of char;
begin
    GetTempPath(MAX_PATH, strPath);
    GetTempFileName(strPath, 'tmp', 0, strFilename);
    Result := StrPas(strFilename);
end;
{$ENDIF}

```

Armed with this knowledge it is quite possible to push the boundaries of your application by writing functions that are specifically tuned for both operating systems.

As you can see, Kylix is an extremely powerful tool for creating software applications that can fully harness the powerful features of the Linux API.

What Windows Has That Linux Doesn't and Vice Versa

There are many things that Windows has that Linux does not. Many of you reading this book are likely coming from a Delphi background, and as a result, you will be familiar with certain Windows technologies that you may like to use.

Some of these are Microsoft-specific features such as COM/DCOM, ADO, and Windows messages. These Windows-specific features are definitely not supported in Kylix. Windows messages are also not used in Kylix, which may be a headache for many developers who rely on a particular functionality that is exposed by Windows messaging, especially those developers who have existing applications that rely on that technology. Other Windows features not available for Kylix developers include using DLL calls that are specific to Windows, such as those found in the Delphi Windows and Shell API units.

Although you will not have access to many of these technologies on Linux, there are many functions that are available under Linux that give you the same or similar functionality. Throughout this book you will see examples of the core Linux API that are similar to Windows API calls.

Now we will get into the fun of the Linux API. In the next several chapters you will discover a vast number of functions you can use in your Linux applications. To begin with, let's look at error management with the Linux API.

Linux API Errors

Dealing with Linux API Errors

The Linux API is not an unreasonable master to serve. It will let you know when an error has occurred within your application. The majority of Linux API calls will give your application the ability to see if any errors occurred. Linux has standards for determining if a function call was successful or not.

The standard rule for the Linux API is if the function call returns an integer value, when the result is negative one (−1), an error occurred; otherwise, the function call was successful. Likewise, if the Linux API call returns a pointer, such as what is used in dealing with streams, if the pointer returns nil then an error also occurred; otherwise, the function was successful.

To retrieve the reason why the function was unsuccessful you use the `errno` function. This function returns an integer value containing one of the constant values found in Table 2.1. A descriptive string can also be obtained by passing in the result of the `errno` into the `strerror` function.

One thing to take into consideration when working with the `errno` function is that the value of `errno` is set after every call to a Linux API call. So, after you call `errno` successfully you would have actually cleared the value to `errno` because the `errno` function was successful. In this way when you work with the errors try to store the return value in a variable and examine the result in that variable.

But the `errno` function is not just for Linux API calls. If you were to create your own shared object (.so) file, you could create your own library of functions, and then if an error occurred within your function you can set the value of `errno` yourself.

As the `errno` is a function, you need to use the `__errno_location` function to retrieve the location of where the `errno` value is stored and then update the value as shown in Listing 2.1. In most cases you would want to set the `errno` value to a value from Table 2.1. It is not advised, but it is possible to make your own error codes outside the range of Table 2.1, by ensuring that the value differs from any value in Table 2.1. If you do choose to create your own error handler and you define your own error code, when the `strerror` function is called, the return of that function will be an unknown error message due to the custom error. As a result, it is wiser to use the standard error codes.



Note: The SysUtils unit also provides several functions for dealing with system error messages for cross-platform applications. You can use the SysError-Code function to retrieve the error code or even use RaiseLastOSError to raise an EOSError exception based on the error code.

Signals as Errors

Signals are often used in the notification of an error to your application. Signals are methods of communicating simple tokens to other processes asynchronously. Be sure to read Chapter 5 to understand the inner workings of how signals are passed between applications.

In regard to managing errors, signals are relevant as a means of error notification because some errors that occur within the kernel or other functions may send a signal to the process instead of or as well as returning the error code to the function. You should be aware that some types of errors will exhibit this behavior.

The most notable example of these errors is EFAULT, which is used when a bad pointer is passed into an application. Under GLibC, the EFAULT message does not get returned by the function, but instead a signal is sent to the process, notifying the application that a segment fault has occurred. Other errors that result in a signal being sent to the process instead of returning from the function include EPIPE, which will return SIGPIPE to your process, and EBACKGROUND, which will return SIGTIN or SIGTTOU. Kylix has been designed to automatically capture signals that deal with errors sent to the process and convert them to exceptions. For more information on dealing with signals see Chapter 5, “Interprocess Communication.”

Linux API Error Functions

errno *LibC.pas*

Syntax

```
function errno: error_t;
```

Description

This function returns the error code of the last system call on the current thread if the Linux API call returned `-1`. Each thread that executes within a process will have its own `errno` stored value so the error code value returned in one thread does not affect the `errno` value in another thread.

Return Value

This function returns the integer error code of the last function code. The function result will be one of the error codes found in Table 2.1.

See Also

`__errno_location`

*Example***Listing 2.1 - Using the `errno` function**

```

if __mkdir('/root/somedir') <> 0 then
begin
  case errno of
    ENOTDIR: showmessage('The file you tried to access could not be created.');
```

```

    EACCES: showmessage('you do not have permissions to create a directory here.');
```

```

  else
    showmessage('Another error has occurred.');
```

```

end;
```

`__errno_location` LibC.pas*Syntax*

```
function __errno_location: PInteger;
```

Description

This function will return the location of the current error value that is returned by the `errno` function. You would use this in any shared object libraries you create to define the type of error that occurs.

Return Value

This function returns a pointer to an integer value, which represents what value will be returned after a call to the `errno` function.

See Also

`errno`

*Example***Listing 2.2 - Using the `__errno_location` function**

```

//To read the error number (same as the errno function)
CurrentErrorNumber := __errno_location^;
```

```

//Set the error number to be returned by errno
__errno_location^ := EEXIST;
```

`perror` LibC.pas*Syntax*

```

procedure perror(
  msg: Pchar
);
```

Description

This function writes a message to standard error, which is normally where output from the console is written, along with the result from a Linux function call. The output to standard

error is the message in the buffer followed by a colon and space, followed by the description of the reason the function was unsuccessful.

Parameters

msg: This is the error message that will be the first portion of the message written out to standard error.

See Also

errno, strerror

Example

Figure 2.1 - Opening a file and examining the perror output



Figure 2.2 - The details of the function when the API function is successful

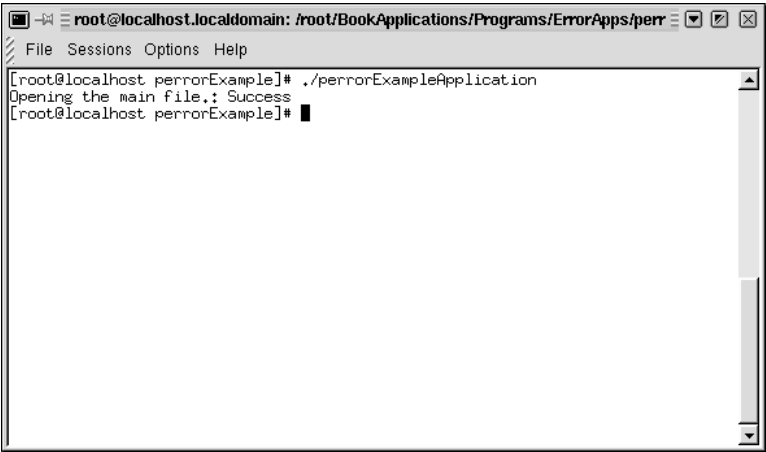
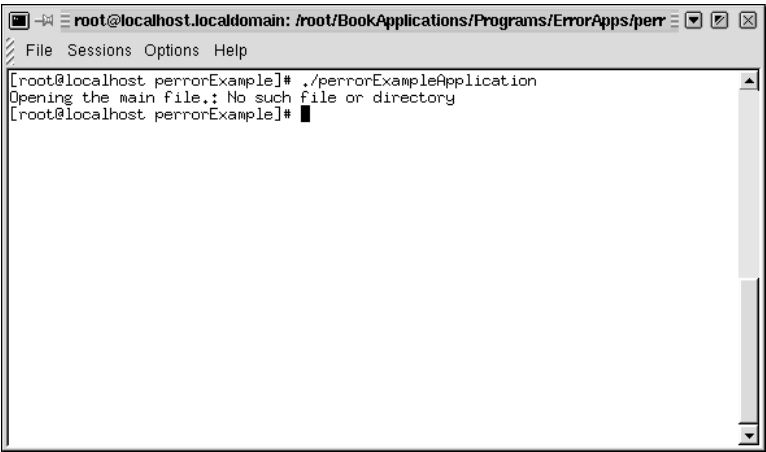


Figure 2.3 - The results from perror when the API function failed



Listing 2.3 - Using the perror function

```

procedure TForm1.Button1Click(Sender: TObject);
var
  f: PIOFile;
begin
  //Open a file for reading
  f := fopen(PChar(edFilename.Text), 'r');

  //write the error out to the standard error
  perror('Opening the main file.');
```

```

  if f <> nil then
    fclose(f);
end;
```

strerror LibC.pas**Syntax**

```

function strerror(
  errcode: Integer
): PChar;
```

Description

This function returns a text message that describes the meaning of a particular Linux API error code.

Parameters

errcode: This is a valid integer error code from one of the values in Table 2.1. If the value passed into this parameter is invalid, the function will return a message saying the error is unknown. An unknown error code may be the result of calling another library that is implementing its own error codes.

Return Value

This function returns a pointer to a text message describing the error. If the error identifier passed into **errcode** is invalid, a message saying that the error is unknown is returned. You should not attempt to modify the string returned from this function as it is managed by the system.

See Also

errno

Example**Listing 2.4 - Using the strerror function**

```

//get the error code
intErrorCode := errno;

Showmessage('The error is '+StrPas(strError(intErrorCode)));
```

Linux API Error Codes

Table 2.1 lists the Linux API error codes.

Table 2.1 - Error codes

Error Code	Error Description
EPERM	The operation cannot be performed because only specific users or processes executing with specific user permissions can perform the operation.
ENOENT	An attempt was made to work on a file or directory that does not exist.
ESRCH	An attempt was made to work on a particular process that does not exist.
EINTR	This error is returned when an asynchronous signal is received, and as a result, the call was interrupted and could not be completed.
EIO	An input/output error occurred while the operation was being performed. This error code is used when access to the device, such as a hard drive or floppy disk, cannot be performed.
ENXIO	This error occurs as a result of trying to access a device, such as a hard drive located at /dev/hd0, that is inaccessible.
E2BIG	When executing an application, the arguments passed to the application take up too much memory.
ENOEXEC	A file was attempted to be executed, but the file is not a valid executable.
EBADF	Bad file descriptor. This can be caused by trying to work on a file descriptor that has not been opened yet or has been closed.
ECHILD	An operation attempted to access or manipulate child processes, but there were no child processes to work with.
EAGAIN, EWOULDBLOCK	The function you tried to perform failed, but the reason it failed was temporary and performing the function again would likely cause the function to succeed.
ENOMEM	The operation failed because the system is out of memory and cannot obtain any more.
EACCES	A file or directory operation was attempted but the calling process did not have permission to perform the operation.
EFAULT	An invalid pointer was accessed by the function. This error code has been superseded under GNU by sending a signal to the process.
ENOTBLK	A block special file, such as /dev/cdrom, is required to perform this operation but instead a regular file was used.
EBUSY	This error is caused by trying to operate on an object or resource that is currently in use and cannot be shared, such as trying to delete a directory when a file in that directory is currently open.
EEXIST	The error is caused by a file or directory that was to be created but already exists.
EXDEV	This was caused by attempting to create a link file on one file system to a file system on another machine. This can also be caused by trying to rename a file to a file on another device.
ENODEV	A device was attempted to be accessed but the device was not the actual device that was required for the function to complete successfully.

Error Code	Error Description
ENOTDIR	A function that works on a directory such as <code>__chdir</code> was attempted on a file.
EISDIR	A directory was accessed for a file operation that is not allowed.
EINVAL	An invalid argument was passed into a parameter that will only accept particular values for that parameter.
ENFILE	The limit on the amount of files for the current process has been reached and the function requires a file descriptor that cannot be created.
EMFILE	This error is returned when the limit on amount of files open on the operating system has been reached.
ENOTTY	A process tried to access terminal input/output on a device or file that is not a terminal.
ETXTBSY	This error is due to attempting to open a file that is currently in use, such as writing to an application file when the file is already open.
EFBIG	This is caused by a file being too large for the system to handle.
ENOSPC	The file operation could not be completed as the disk has reached its capacity size.
ESPIPE	This is caused by using a <code>seek</code> , <code>fseek</code> , or <code>lseek</code> function on a file descriptor that does not allow seeking, such as a pipe or FIFO.
EROFS	This error was caused by an attempt to write to a read-only file system such as a CD-ROM.
EMLINK	Too many links already exist for this file.
EPIPE	This error is caused by writing into a pipe but not having another process read from the pipe.
EDOM	An argument is out of the domain of the function.
ERANGE	The result of a math operation is not in a displayable range due to an overflow or underflow operation.
EDEADLK	Kernel error: Resource deadlock would occur
ENAMETOOLONG	This error is caused when the filename is too long. This error usually occurs when you are creating or renaming a file.
ENOLCK	There are no more locks available. This functionality will not occur under Glibc, but may occur as a result of connecting to a remote device that may be running under UNIX or another operating system.
ENOSYS	The function that was called does not exist in the Glibc library. This could be caused by having a previous version of Glibc, in which case you would need to update your version of the library.
ENOTEMPTY	This error occurs when a directory has files in it and the operation requires that the directory be empty. This is usually as a result of attempting to remove a directory.
ELOOP	The file or directory that you tried to open was actually a link file or directory, which resulted in calculating nested links. The error occurred because the maximum amount of links to traverse had been reached.
ENOMSG	No message of desired type.
EIDRM	Identifier has been removed.
ECHRNG	Kernel error: Channel number out of range
EL2NSYNC	Kernel error: Level 2, not synchronized error

Error Code	Error Description
EL3HLT	Kernel error: Level 3, halted
EL3RST	Kernel error: Level 3, reset
ELNRNG	Kernel error: Link number out of range
EUNATCH	Kernel error: Protocol driver not attached
ENOCSS	Kernel error: No CSI structure available
EL2HLT	Kernel error: Level 2, halted
EBADE	Kernel error: Invalid exchange
EBADR	Kernel error: Invalid request descriptor
EXFULL	Kernel error: Exchange full
ENOANO	Kernel error: No anode
EBADRQC	Kernel error: Invalid request code
EBADSLT	Kernel error: Invalid slot
EBFONT	Kernel error: Bad font file format
ENOSTR	The device is not a stream.
ENODATA	No data available.
ETIME	A stream timeout error occurred using ioctl.
ENOSR	Out of streams resources.
ENONET	Kernel error: Machine is not on the network
ENOPKG	Kernel error: Package not installed
EREMOTE	This error was caused by a failed attempt to mount a remote device to a device name that was already mounted.
ENOLINK	The link has been severed.
EADV	Kernel error: Advertise error
ESRMNT	Kernel error: Srmount error
ECOMM	Kernel error: Communication error on send
EPROTO	Protocol error.
EMULTIHOP	Multihop attempted.
EDOTDOT	Kernel error: RFS-specific error
EBADMSG	Not a data message.
EOVERFLOW	The error was caused due to a value being too large for the defined data type.
ENOTUNIQ	Kernel error: Name not unique on network
EBADFD	Kernel error: File descriptor in bad state
EREMCHG	Kernel error: Remote address changed
ELIBACC	Kernel error: Cannot access a needed shared library
ELIBBAD	Kernel error: Accessing a corrupted shared library
ELIBSCN	Kernel error: .lib section in an executable application corrupted
ELIBMAX	Kernel error: Attempting to link in too many shared libraries
ELIBEXEC	Kernel error: Cannot exec a shared library directly
EILSEQ	Caused by an illegal byte sequence.
ERESTART	Interrupted system call should be restarted.
ESTRPIPE	Streams pipe error.

Error Code	Error Description
EUSERS	This error is caused by the system not being able to handle the number of users currently logged onto the system.
ENOTSOCK	A file operation on a socket was required for the function, but the parameter was not a socket.
EDESTADDRREQ	No destination address was set for the socket but was required for the function to work correctly. You will most likely see this error when creating socket functions.
EMSGSIZE	A message was to be sent to the socket in one atomic unit, but the message was too large.
EPROTOTYPE	The protocol requested for the function does not match the type of socket. An example of this is using a TCP function that is only applicable to a UDP socket.
ENOPROTOOPT	The protocol that you requested could not be created. This means that you are trying to call a function with the wrong socket type as in EPROTOTYPE, or that the library does not know how to handle the parameters that you have passed into the function.
EPROTONOSUPPORT	The network protocol that you want to create is not supported by GLibC. You may run across this if the protocol does not exist or is not implemented in the version of GLibC that you have running.
ESOCKTNOSUPPORT	The socket type that you have requested is not supported by GLibC either because the socket type has not been implemented or the socket type does not exist.
EOPNOTSUPP	The operation is not supported in the socket function that you are calling.
EPFNOSUPPORT	The socket protocol family is not supported in this version of GLibC.
EAFNOSUPPORT	The address family is not supported by the socket protocol.
EADDRINUSE	The socket that you are requesting is already in use. This may be due to having a server listening on a port in one process and attempting to have another process listen on the same port.
EADDRNOTAVAIL	You have tried to create a socket address that you cannot access. For example you may have attempted to create a server socket that listens on an IP address that is not configured for your machine.
ENETDOWN	The function failed due to the network being down.
ENETUNREACH	The particular function failed because the network address was not reachable. A client socket may cause this error when it requests to connect to a server which is outside of its subnet and there is no default gateway.
ENETRESET	The network dropped the connection due to the remote server losing its network or the machine was reset.
ECONNABORTED	The connection was aborted by the software.
ECONNRESET	The socket you were connected to was reset.
ENOBUFS	This error occurs when the buffers used to handle input/output communication become full, so the read/write operation fails.
EISCONN	This error is caused by trying to connect to a socket that is already connected. This is often caused by calling connect twice.
ENOTCONN	You are trying to perform a socket operation, such as reading or writing, to a socket that is not connected.

Error Code	Error Description
ESHUTDOWN	This is caused by an attempt to perform a socket operation on a socket that has already shut down.
ETOOMANYREFS	Too many references; cannot splice.
ETIMEDOUT	This error was caused by not being able to connect to the socket before the timeout occurred.
ECONNREFUSED	The connection to the remote server was refused. This could be due to IP address restrictions from your host machine.
EHOSTDOWN	The host connection is down. This is usually because the network connection cannot be reached.
EHOSTUNREACH	The host server cannot be reached on the network.
EALREADY	Operation already in progress.
EINPROGRESS	Operation now in progress.
ESTALE	A file handle to a network file system has been removed but the device has since changed. An example of this is a mounted network connection to a hard drive that disconnected after a handle has already been obtained.
EUCLEAN	Kernel error: Structure needs cleaning
ENOTNAM	Kernel error: Not a XENIX named type file
ENAVAIL	Kernel error: No XENIX semaphores available
EISNAM	Kernel error: Is a named type file
EREMOTEIO	Kernel error: Remote I/O error
EDQUOT	The error was caused due to an operation that performs a disk operation resulting in the user's disk quota being reached. As a result, the function could not continue.
ENOMEDIUM	Kernel error: No medium found
EMEDIUMTYPE	Kernel error: Wrong medium type

Errors are Not Exceptions

If you have been using Delphi or Kylix for any length of time, you will know that the default behavior for dealing with any error is to wrap your code block in either a try..finally or try..except block. Both the VCL and CLX make heavy use of exceptions throughout their function calls, which gives a consistent method for handling any problems.

Unfortunately, the Linux API uses integer and pointer result values to denote that an error has occurred and then sets the error code to describe what kind of error occurred. You should check the return value of a function in many cases to actually examine the error, as demonstrated in Listing 2.5.



Note: Windows API programmers should be already used to this convention of dealing with errors. The vast majority of Windows API functions return error codes such as false, nil, or -1, and the last error code can be retrieved using the GetLastError function.

Listing 2.5 - Working with Linux API error trapping

```

Uses
    LibC;

Procedure TForm1.btnCreateDirectoryClick(Sender: TObject);
var
    ecode: integer;
begin
    ecode := __mkdir('/root/newdir');
    case ecode of
        0: ShowMessage('The directory was created.');
```

EACCES: ShowMessage('You do not have correct permissions '+
 'to create this directory.');

```

    end;
end;
```

In Listing 2.5, you would use the error codes in Table 2.1 within your applications. However, as these errors will not raise exceptions, Listing 2.7 provides a unit that will process failed function calls as exceptions so that you can trap them using conventional Kylix coding standards.

The unit itself is designed to be flexible so that if you choose to implement more descriptive text descriptions for the error messages you should be able to do so. Likewise, if you are converting another library of functions, you may find that you need to implement an additional version of the LibCCheck function to implement a new data type.

Listing 2.6 - Using a Linux API error-to-exception conversion unit

```

Uses
    LibCExceptions;

Procedure TForm1.btnCreateDirectoryClick(Sender: TObject);
begin
    Try
        LibCCheck(__mkdir('/root/newdir'));
        MessageDlg('The directory was created.', mtInformation,
            [mbOk], 0);
    except
        on E: ELibCException do
            MessageDlg(E.Message, mtWarning, [mbOk], 0);
    end;
end;
```

Listing 2.7 - LibCExceptions.pas

```

unit LibCExceptions;

{
    LibCExceptions.pas

    Written for "The Tomes of Kylix - Linux API"
    for Wordware Publishing
    by Glenn Stephens (glenn@coderage.com)
```

```

        This unit is used to wrap any LibC function and
        convert any error that occurs to map to a valid
        exception.
    }

interface

uses
    SysUtils, Types, Classes, LibC;

type
    //The Main LibC Exception
    ELibCException = class(Exception);

    //Other main LibC Error types as exceptions
    ELibCIOException = class(ELibCException);
    ELibCThreadException = class(ELibCException);
    ELibCProcessException = class(ELibCException);
    ELibCSignalException = class(ELibCException);
    ELibCPipeException = class(ELibCException);
    ELibCNetworkException = class(ELibCException);
    ELibCSocketException = class(ELibCNetworkException);
    ELibCKernelException = class(ELibCException);
    ELibCSecurityException = class(ELibCException);
    ELibCMathException = class(ELibCException);
    ELibCSharedObjectException = class(ELibCException);

    //The entire LibC errors wrapped
    ELibCEPERMException = class(ELibCSecurityException);
    ELibCENOENTException = class(ELibCIOException);
    ELibCESRCHException = class(ELibCProcessException);
    ELibCEINTRException = class(ELibCSignalException);
    ELibCEIOException = class(ELibCIOException);
    ELibCENXIOException = class(ELibCIOException);
    ELibCE2BIGException = class(ELibCProcessException);
    ELibCENOEXECException = class(ELibCProcessException);
    ELibCEBADFException = class(ELibCIOException);
    ELibCECHILDEException = class(ELibCProcessException);
    ELibCEAGAINException = class(ELibCException);
    ELibCENOMEMException = class(ELibCException);
    ELibCEACCSEException = class(ELibCSecurityException);
    ELibCEFAULTException = class(ELibCException);
    ELibCENOTBLKException = class(ELibCIOException);
    ELibCEBUSYException = class(ELibCException);
    ELibCEEXISTException = class(ELibCIOException);
    ELibCEXDEVException = class(ELibCIOException);
    ELibCENODEVException = class(ELibCIOException);
    ELibCENOTDIRException = class(ELibCIOException);
    ELibCEISDIRException = class(ELibCIOException);
    ELibCEINVALException = class(ELibCException);
    ELibCENFILEException = class(ELibCIOException);
    ELibCEMFILEException = class(ELibCIOException);
    ELibCENOTTYException = class(ELibCIOException);
    ELibCETXTBSYException = class(ELibCIOException);
    ELibCEFBIGException = class(ELibCIOException);
    ELibCENOSPCException = class(ELibCIOException);
    ELibCESPIPEException = class(ELibCIOException);

```

```

ELibCEROFSEException = class(ELibCIOException);
ELibCEMLINKException = class(ELibCIOException);
ELibCEPIPEException = class(ELibCIOException);
ELibCEDOMException = class(ELibCMathException);
ELibCERANGEException = class(ELibCMathException);
ELibCEDEADLKEException = class(ELibCException);
ELibCENAMETOOLONGException = class(ELibCKernelException);
ELibCENOLCKException = class(ELibCIOException);
ELibCENOSYSEException = class(ELibCException);
ELibCENOTEMPTYException = class(ELibCIOException);
ELibCELOOPException = class(ELibCIOException);
ELibCENOMSGException = class(ELibCException);
ELibCEIDRMException = class(ELibCException);
ELibCECHRNGEException = class(ELibCKernelException);
ELibCEL2NSYNCEException = class(ELibCKernelException);
ELibCEL3HLTException = class(ELibCKernelException);
ELibCEL3RSTException = class(ELibCKernelException);
ELibCELNRNGEException = class(ELibCKernelException);
ELibCEUNATCHException = class(ELibCKernelException);
ELibCENOCSEException = class(ELibCKernelException);
ELibCEL2HLTException = class(ELibCKernelException);
ELibCEBADEException = class(ELibCKernelException);
ELibCEBADREException = class(ELibCKernelException);
ELibCEXFULLEException = class(ELibCKernelException);
ELibCENOANOException = class(ELibCKernelException);
ELibCEBADRQCEException = class(ELibCKernelException);
ELibCEBADSLTException = class(ELibCKernelException);
ELibCEBFONTEException = class(ELibCKernelException);
ELibCENOSTREException = class(ELibCIOException);
ELibCENODATAException = class(ELibCException);
ELibCETIMEEException = class(ELibCException);
ELibCENOSREException = class(ELibCIOException);
ELibCENONETException = class(ELibCKernelException);
ELibCENOPKGEException = class(ELibCKernelException);
ELibCEREMOTEException = class(ELibCIOException);
ELibCENOLINKException = class(ELibCException);
ELibCEADVEException = class(ELibCKernelException);
ELibCESRMNTEException = class(ELibCKernelException);
ELibCECOMMEException = class(ELibCKernelException);
ELibCEPROTOEException = class(ELibCException);
ELibCEMULTIHOPException = class(ELibCException);
ELibCEDOTDOTException = class(ELibCKernelException);
ELibCEBADMSGException = class(ELibCException);
ELibCEOVERFLOWException = class(ELibCException);
ELibCENOTUNIQUEException = class(ELibCKernelException);
ELibCEBADFDEException = class(ELibCKernelException);
ELibCEREMCHGException = class(ELibCKernelException);
ELibCELIBACCEException = class(ELibCKernelException);
ELibCELIBBADEException = class(ELibCKernelException);
ELibCELIBSCNException = class(ELibCKernelException);
ELibCELIBMAXException = class(ELibCKernelException);
ELibCELIBEXECEException = class(ELibCKernelException);
ELibCEILSEQException = class(ELibCException);
ELibCERESTARTEException = class(ELibCException);
ELibCESTRPIPEException = class(ELibCIOException);
ELibCEUSERSEException = class(ELibCIOException);
ELibCENOTSOCKException = class(ELibCSocketException);

```

```

ELibCEDESTADDRREQException = class(ELibCSocketException);
ELibCEMSGSIZEException = class(ELibCSocketException);
ELibCEPROTOTYPEException = class(ELibCSocketException);
ELibCENOPROTOPTException = class(ELibCSocketException);
ELibCEPROTONOSUPPORTException = class(ELibCSocketException);
ELibCESOCKTNOSUPPORTException = class(ELibCSocketException);
ELibCEOPNOTSUPPException = class(ELibCSocketException);
ELibCEPFNOSUPPORTException = class(ELibCSocketException);
ELibCEAFNOSUPPORTException = class(ELibCSocketException);
ELibCEADDRINUSEException = class(ELibCSocketException);
ELibCEADDRNOTAVAILException = class(ELibCSocketException);
ELibCENETDOWNException = class(ELibCSocketException);
ELibCENETUNREACHException = class(ELibCSocketException);
ELibCENETRESETException = class(ELibCSocketException);
ELibCECONNABORTEDEException = class(ELibCSocketException);
ELibCECONNRESETException = class(ELibCSocketException);
ELibCENOBUFXException = class(ELibCIOException);
ELibCEISCONNException = class(ELibCSocketException);
ELibCENOTCONNException = class(ELibCSocketException);
ELibCESHUTDOWNException = class(ELibCSocketException);
ELibCETOOMANYREFSException = class(ELibCException);
ELibCETIMEDOUTException = class(ELibCSocketException);
ELibCECONNREFUSEDException = class(ELibCSocketException);
ELibCEHOSTDOWNException = class(ELibCSocketException);
ELibCEHOSTUNREACHException = class(ELibCSocketException);
ELibCEALREADYException = class(ELibCException);
ELibCEINPROGRESSEXception = class(ELibCException);
ELibCESTALEXception = class(ELibCIOException);
ELibCEUCLEANException = class(ELibCKernelException);
ELibCENOTNAMException = class(ELibCKernelException);
ELibCENAVAILException = class(ELibCKernelException);
ELibCEISNAMException = class(ELibCKernelException);
ELibCEREMOTEIOException = class(ELibCKernelException);
ELibCEDQUOTException = class(ELibCIOException);
ELibCENOMEDIUMException = class(ELibCKernelException);
ELibCEMEDIUMTYPEException = class(ELibCKernelException);

//Overloaded functions to handle converting Linux API
//error to Kylix Exceptions.

function LibCCheck(Value: integer): integer; overload;
function LibCCheck(Value: Pointer): Pointer; overload;

function GetLibCErrorMessage(intErrNo: integer): string;

implementation

procedure ProcessLibCError(intErrNo: integer);
var
  errStr: string;
begin
  //This function converts the error number to a valid
  //exception. By default the error will be the standard
  //error from Linux. You should change the Message for
  //each function to reflect how you would like errors
  //displayed to your users.

```

```

errStr := GetLibCErrorMessage(intErrNo);

case intErrNo of
    EPERM: raise ELibCEPERMException.Create(errStr);
    ENOENT: raise ELibCENOENTException.Create(errStr);
    ESRCH: raise ELibCESRCHException.Create(errStr);
    EINTR: raise ELibCEINTRException.Create(errStr);
    EIO: raise ELibCEIOException.Create(errStr);
    ENXIO: raise ELibCENXIOException.Create(errStr);
    E2BIG: raise ELibCE2BIGException.Create(errStr);
    ENOEXEC: raise ELibCENOEXECException.Create(errStr);
    EBADF: raise ELibCEBADFException.Create(errStr);
    ECHILD: raise ELibCECHILDException.Create(errStr);
    EAGAIN: raise ELibCEAGAINException.Create(errStr);
    ENOMEM: raise ELibCENOMEMException.Create(errStr);
    EACCES: raise ELibCEACCSException.Create(errStr);
    EFAULT: raise ELibCEFAULTException.Create(errStr);
    ENOTBLK: raise ELibCENOTBLKException.Create(errStr);
    EBUSY: raise ELibCEBUSYException.Create(errStr);
    EEXIST: raise ELibCEEXISTException.Create(errStr);
    EXDEV: raise ELibCEXDEVException.Create(errStr);
    ENODEV: raise ELibCENODEVException.Create(errStr);
    ENOTDIR: raise ELibCENOTDIRException.Create(errStr);
    EISDIR: raise ELibCEISDIRException.Create(errStr);
    EINVAL: raise ELibCEINVALException.Create(errStr);
    ENFILE: raise ELibCENFILEException.Create(errStr);
    EMFILE: raise ELibCEMFILEException.Create(errStr);
    ENOTTY: raise ELibCENOTTYException.Create(errStr);
    ETXTBSY: raise ELibCETXTBSYException.Create(errStr);
    EFBIG: raise ELibCEFBIGException.Create(errStr);
    ENOSPC: raise ELibCENOSPCException.Create(errStr);
    ESPIPE: raise ELibCESPIPEException.Create(errStr);
    EROFS: raise ELibCEROFSException.Create(errStr);
    EMLINK: raise ELibCEMLINKException.Create(errStr);
    EPIPE: raise ELibCEPIPEException.Create(errStr);
    EDOM: raise ELibCEDOMException.Create(errStr);
    ERANGE: raise ELibCERANGEException.Create(errStr);
    EDEADLK: raise ELibCEDEADLKException.Create(errStr);
    ENAMETOOLONG: raise ELibCENAMETOOLONGException.Create(errStr);
    ENOLCK: raise ELibCENOLCKException.Create(errStr);
    ENOSYS: raise ELibCENOSYSException.Create(errStr);
    ENOTEMPTY: raise ELibCENOTEMPTYException.Create(errStr);
    ELOOP: raise ELibCELOOPException.Create(errStr);
    ENOMSG: raise ELibCENOMSGException.Create(errStr);
    EIDRM: raise ELibCEIDRMException.Create(errStr);
    ECHRNG: raise ELibCECHNRNGException.Create(errStr);
    EL2NSYNC: raise ELibCEL2NSYNCException.Create(errStr);
    EL3HLT: raise ELibCEL3HLTException.Create(errStr);
    EL3RST: raise ELibCEL3RSTException.Create(errStr);
    ELNRNG: raise ELibCELNRNGException.Create(errStr);
    EUNATCH: raise ELibCEUNATCHException.Create(errStr);
    ENOCSI: raise ELibCENOCSException.Create(errStr);
    EL2HLT: raise ELibCEL2HLTException.Create(errStr);
    EBADE: raise ELibCEBADEException.Create(errStr);
    EBADR: raise ELibCEBADRException.Create(errStr);
    EXFULL: raise ELibCEXFULLException.Create(errStr);
    ENOANO: raise ELibCENOANOException.Create(errStr);

```

```

EBADRQC: raise ELibCEBADRQCException.Create(errStr);
EBADSLT: raise ELibCEBADSLTException.Create(errStr);
EBFONT: raise ELibCEBFONTEException.Create(errStr);
ENOSTR: raise ELibCENOSTRException.Create(errStr);
ENODATA: raise ELibCENODATAException.Create(errStr);
ETIME: raise ELibCETIMEException.Create(errStr);
ENOSR: raise ELibCENOSRException.Create(errStr);
ENONET: raise ELibCENONETException.Create(errStr);
ENOPKG: raise ELibCENOPKGException.Create(errStr);
EREMOTE: raise ELibCEREMOTEXception.Create(errStr);
ENOLINK: raise ELibCENOLINKEException.Create(errStr);
EADV: raise ELibCEADVEException.Create(errStr);
ESRMNT: raise ELibCESRMNTEException.Create(errStr);
ECOMM: raise ELibCECOMMEException.Create(errStr);
EPROTO: raise ELibCEPROTOException.Create(errStr);
EMULTIHOP: raise ELibCEMULTIHOPException.Create(errStr);
EDOTDOT: raise ELibCEDOTDOTException.Create(errStr);
EBADMSG: raise ELibCEBADMSGException.Create(errStr);
EOVERFLOW: raise ELibCEOVERFLOWException.Create(errStr);
ENOTUNIQ: raise ELibCENOTUNIQException.Create(errStr);
EBADFD: raise ELibCEBADFDEException.Create(errStr);
EREMCHG: raise ELibCEREMCHGException.Create(errStr);
ELIBACC: raise ELibCELIBACCException.Create(errStr);
ELIBBAD: raise ELibCELIBBADException.Create(errStr);
ELIBSCN: raise ELibCELIBSCNException.Create(errStr);
ELIBMAX: raise ELibCELIBMAXException.Create(errStr);
ELIBEXEC: raise ELibCELIBEXECException.Create(errStr);
ETLSEQ: raise ELibCEILSEQException.Create(errStr);
ERESTART: raise ELibCERESTARTEException.Create(errStr);
ESTRPIPE: raise ELibCESTRPIPEException.Create(errStr);
EUSERS: raise ELibCEUSERSEException.Create(errStr);
ENOTSOCK: raise ELibCENOTSOCKException.Create(errStr);
EDESTADDRREQ: raise ELibCEDESTADDRREQException.Create(errStr);
EMSGSIZE: raise ELibCEMSGSIZEException.Create(errStr);
EPROTOTYPE: raise ELibCEPROTOTYPEException.Create(errStr);
ENOPROTOOPT: raise ELibCENOPROTOOPTException.Create(errStr);
EPROTONOSUPPORT: raise ELibCEPROTONOSUPPORTException.Create(errStr);
ESOCKTNOSUPPORT: raise ELibCESOCKTNOSUPPORTException.Create(errStr);
EOPNOTSUPP: raise ELibCEOPNOTSUPPEXception.Create(errStr);
EPFNOSUPPORT: raise ELibCEPFNOSUPPORTException.Create(errStr);
EAFNOSUPPORT: raise ELibCEAFNOSUPPORTException.Create(errStr);
EADDRINUSE: raise ELibCEADDRINUSEException.Create(errStr);
EADDRNOTAVAIL: raise ELibCEADDRNOTAVAILException.Create(errStr);
ENETDOWN: raise ELibCENETDOWNException.Create(errStr);
ENETUNREACH: raise ELibCENETUNREACHEXception.Create(errStr);
ENETRESET: raise ELibCENETRESETException.Create(errStr);
ECONNABORTED: raise ELibCECONNABORTEDEXception.Create(errStr);
ECONNRESET: raise ELibCECONNRESETException.Create(errStr);
ENOBUFS: raise ELibCENOBUFSEException.Create(errStr);
ETSCONN: raise ELibCEISCONNException.Create(errStr);
ENOTCONN: raise ELibCENOTCONNException.Create(errStr);
ESHUTDOWN: raise ELibCESHUTDOWNException.Create(errStr);
ETOOMANYREFS: raise ELibCETOOMANYREFSEException.Create(errStr);
ETIMEDOUT: raise ELibCETIMEDOUTException.Create(errStr);
ECONNREFUSED: raise ELibCECONNREFUSEDEXception.Create(errStr);
EHOSTDOWN: raise ELibCEHOSTDOWNException.Create(errStr);
EHOSTUNREACH: raise ELibCEHOSTUNREACHEXception.Create(errStr);

```

```

EALREADY: raise ELibCEALREADYException.Create(errStr);
ENPROGRESS: raise ELibCEINPROGRESSEException.Create(errStr);
ESTALE: raise ELibCESTALEException.Create(errStr);
EUCLEAN: raise ELibCEUCLEANException.Create(errStr);
ENOTNAM: raise ELibCENOTNAMEException.Create(errStr);
ENAVAIL: raise ELibCENAVAILException.Create(errStr);
EISNAM: raise ELibCEISNAMEException.Create(errStr);
EREMOTEIO: raise ELibCEREMOTEIOException.Create(errStr);
EDQUOT: raise ELibCEDQUOTException.Create(errStr);
ENOMEDIUM: raise ELibCENOMEDIUMException.Create(errStr);
EMEDIUMTYPE: raise ELibCEMEDIUMTYPEException.Create(errStr);
else
begin
    //Return the default message for the error code.
    raise ELibCException.Create(errStr);
end;
end;
end;

function LibCCheck(Value: integer): integer; overload;
begin
    //This is the integer version of the function that is returned
    Result := Value;

    if Result = -1 then
    begin
        //We have an error
        ProcessLibCError(errno);
    end;
end;

function LibCCheck(Value: Pointer): Pointer; overload;
begin
    //This is the pointer version of the function which is used
    //in many functions such as Stream functions such as fopen etc.
    Result := Value;

    if Result = nil then
    begin
        //We have an error
        ProcessLibCError(errno);
    end;
end;

function GetLibCErrorMessage(intErrNo: integer): string;
var
    pErrorMsg: PChar;
begin
    //Generic function that wraps the 'strerror' function to
    //get a Kylix string
    pErrorMsg := strerror(intErrNo);
    Result := StrPas(pErrorMsg);
end;

end.

```


Conclusion

Error management under the Linux API is quite solid and thorough, but while running any application under the Linux API, it is essential to cover any error situation that may occur. The techniques in this chapter will assist you in managing the error whether you are using the normal method of examining the results of each function call or choose to convert the API errors to Kylix exceptions.

In addition to the techniques in this chapter, there are also several special cases for error handling that are not covered, such as working with errors on multi-threaded applications and dealing with errors when your application is running as a daemon. These techniques will be discussed in the chapters on threads and processes.

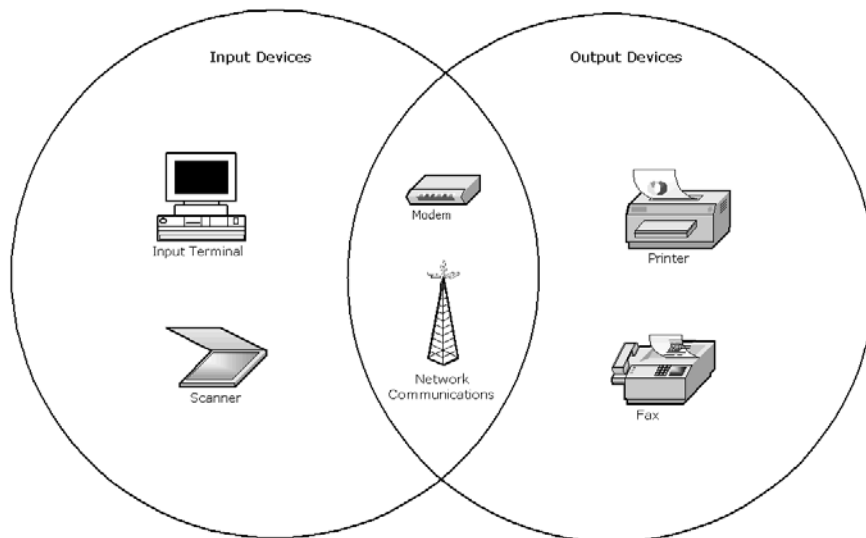
Linux Input and Output

Introduction

One of the most common tasks when developing any piece of software is reading and writing to files. No matter what you do, you are bound to interact with a file in some way. When you want to load some settings, those settings are most likely stored within a file. When you want to write some records, those records are most likely stored within a file. Most documents you use in software such as word processors will be stored as files, and the majority of software applications today will have to deal with files in one way or another.

Linux is no exception to this reliance on files, but it also extends the concept of a file. In Linux, almost every interaction to any device is through a file. Typing on the keyboard, writing to a terminal window, and even manipulating the sound card is done by communicating with a file. This is because Linux uses files as a form of polymorphic interface to input and output devices. In the later chapters of this book you will find numerous areas that reference reading and writing data from a device or communication, in exactly the same way communication is done via files.

*Figure 3.1 -
Some of the
many
input/output
methods
under Linux*



In this chapter you will see the basics of working with files using the available Linux API calls and become familiar with the many conventions of Linux input and output. Along the way you will also discover some neat little tips and tricks for working with files, such as mixing Kylix and Linux file types, analyzing files and directory information, the basics of managing Linux permissions through code, and more.

You will also find a fantastic reference to many of the Linux API calls that correspond to Linux input and output. To top this off there is source code to a TClientDataSet descendant that can load a simple dBase file using techniques covered in this chapter.

So let's begin with the different ways you can communicate with files on Linux: the file descriptor and the stream.



Note: Files under Linux If you have come from a Windows background, the first thing that you will notice is that there is no concept of drives. All files are referenced as either having the full path of the file or the relative path of the filename. A fully defined path name starts with the forward slash symbol and contains the directory name and the filename, for example, `/etc/kylix/readme.txt`.

A relative path will contain either the filename only (`readme.txt`) or the directory and the filename (`kylix/readme.txt`).

These filenames assume that your current working directory is set to something where it could find the file. In the case where the file to be loaded was `readme.txt`, your application would need to have its current directory set to a directory where the `readme.txt` file exists, such as `/home/glenn/kylix/`, to load the file `/home/glenn/kylix/readme.txt`. Likewise, if the directory was included in the relative path, the application will look within the subdirectory when loading the file. So to load the file `kylix/readme.txt`, the current directory would need to be `/home/glenn/`.

File Descriptors and Streams

Under Linux, there are two main ways to perform an operation on a file. The first is using a low-level interface known as a file descriptor.

A file descriptor is an integer handle that is returned by a call to the Linux API open function as shown in Listing 3.1.

While working with file descriptors can be fast and efficient, there are only a limited number of functions that are available for reading and writing data from a file descriptor. The alternative is to use another kind of input/output method, the stream.

Using streams under Linux gives you as the developer much more functionality to work with when operating with files. Streams are designed to be a higher level interface than a file descriptor. Streams provide such functionality as reading in lines of text, reading until a delimiter is obtained, and working with blocks of memory of various sizes. A stream is normally returned by a call to the `fopen` function as shown in Listing 3.2.



Note: Do not confuse the Linux API stream with CLX's TStream class. A stream under Linux is a low-level interface that refers to a pointer to a TIOFile record that is then used as a handle for many input and output operations. TStream is similar in purpose, but is specific to the CLX library. Throughout the rest of this chapter, the term “stream” will refer to a Linux API stream.

Now that you know of the two ways to work with files, let's examine the techniques for working with files under both circumstances.

Input and Output Methods

Managing input and output within your application is quite an easy process. You open or create the file, read or write the information to the file, and then close the file. There are various ways that you can do this with the Linux API, and you need to make sure you choose the best method for your needs.

Opening and Closing Files

Opening a file is a straightforward process. There are a number of methods for opening a file. When using a file descriptor, you use the open function shown in Listing 3.1. Alternatively, fopen is used to open a file when using a stream, as shown in Listing 3.2.

Listing 3.1 - Opening a file descriptor

```
Var
    FileHandle: integer;
begin
    FileHandle := open(PChar(edFilename.Text), O_RDWR, 0);

    //Perform some operations on the file

    __close(FileHandle);
end;
```

When working with file descriptors, the open function will take three parameters. The first parameter is a path to the file that will be opened.

The second parameter is a constant value that specifies if the file is going to be opened for reading (O_RDONLY), writing (O_WRONLY), or both (O_RDWR), and whether the file will be created if needed (O_CREAT) or if the file is going to be opened exclusively (O_EXCL). Other available options for this parameter, are listed in Table 3.8.

The last parameter for the open function specifies the permissions that will be used for the file. File and directory permissions are covered in the section titled “File Permissions and Ownership” later in this chapter. Setting a value of zero for this parameter will use the default permissions for the application.

If all the information in the open function is correct, the open function will return an integer value that represents a file. This integer value is the file descriptor.

Listing 3.2 - Opening a stream

```
Var
  StreamFilePointer: PIOFile;
begin
  StreamFilePointer := fopen(PChar(edFilename.Text), 'w+');

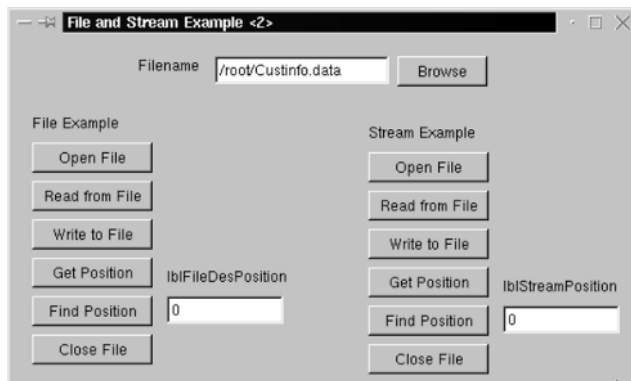
  //Perform operations on the file stream.

  fclose(StreamFilePointer);
end;
```

When working with a stream, opening a file is done using the fopen function. Like the open function, the fopen function's first parameter is the name of the file that is to be opened. The second parameter for the fopen function specifies how the function is to be opened (for reading or writing, etc.). This parameter is actually a null-terminated string that will normally either be the value "r" for read-only, "w" for writing, "r+" or "w+" for both read and write, or "a" for appending to a file.

If all the information is correct for the fopen function, the function will return a valid pointer to a TIOFile. It is this pointer that is the stream.

Figure 3.2 -
Simple
operations on
files and
streams



Reading and Writing Files

Like opening and closing files, methods of reading and writing files depends on whether you are using file descriptors or streams. The Linux API functions allow reading and writing of single characters, strings, and blocks of data from file. What is more evident is that there are many more functions for reading and writing using streams than there are for using file descriptors. This is due to the fact that file descriptors were meant to be the low-level interface to files, whereas streams were designed to be an abstract way of dealing with files so you are not complicated with working with the limited functionality of file descriptors.

Although the techniques you will see for working with Linux files will work, it may not be in your application’s best interest to use file descriptors or streams. Kylix’s file handling, which uses traditional Pascal types such as File and TextFile, is usually sufficient for most purposes. By using the native Pascal types, you will also get the benefit of having cross-platform code.

The most benefit would be gained from using file descriptors or file streams when you really need to push the performance of the application, such as when you are running within a high-volume server process or an embedded application.

Table 3.1 lists the different API calls for reading and writing with both file descriptors and file streams.

Table 3.1 - File operations using various methods

Operation	API for File Descriptors	API for File Streams	Normal Pascal Method
Reading a single character	__read	getc, fgetc	read
Writing a single character	__write	putc, fputc	write
Reading a string	__read	fgets, getline	readln
Writing a string	__write	fputs	write
Reading a block of data	__read	fread	BlockRead
Writing a block of data	__write	fwrite	BlockWrite

As you can see from Table 3.1, when working with file descriptors, the only real functions for reading and writing data are the __read and __write functions. Listing 3.3 demonstrates reading from a file descriptor using the __read API call. The __read function takes several parameters. The first parameter is the file descriptor that will be read from. The second parameter is the buffer where the data that will be read is stored, and finally, the last parameter specifies how many bytes should be read from the file descriptor.

Listing 3.3 - Reading from a file descriptor

```
var
  SmallStr: array[0..4] of char;
begin
  //Read a block of 5 characters from the file at a time
  //This does not test for the End of File
  __read(FileHandle, SmallStr, 5);

  Caption := SmallStr;
end;
```

Writing to a file descriptor is very similar to reading from a file descriptor. Using the __write Linux API function, the first parameter specifies the file descriptor that will be written to, the second parameter specifies the buffer that holds the content that will be written to the file descriptor, and the third parameter specifies how many bytes from the buffer will be written to the file descriptor. A demonstration of writing to a file descriptor is shown in Listing 3.4.

Listing 3.4 - Writing to a file descriptor

```

var
    SmallStr: array[0..4] of char;
begin
    //Writes a block of 5 characters to the file at a time
    SmallStr := 'abcde';
    __write(FileHandle, SmallStr, 5);
end;

```

As you can see, working with file descriptors to read and write data is extremely simple using the `__read` and `__write` functions. But these are not powerful enough for most programmers. Using streams there are many more available options for reading and writing to a file, such as reading an individual line from the file, reading or writing a single character, or reading until a delimiter character is reached.

For example, reading and writing from a stream can be done in much the same way as a file descriptor, where the data is written into a buffer. When using streams, however, the `fread` and `fwrite` functions are used. These functions both take four parameters: the buffer that is used, the size of the block of data to read or write, how many blocks should be read or written, and finally the stream that will be used. The main difference between `fread` and `fwrite` is that `fread` is used for reading from the stream as shown in Listing 3.5, and `fwrite` is used for writing to the stream as seen in Listing 3.6.

Working with streams can be quite powerful and there are numerous functions that are useful in reading data from a stream, such as `fgets` for reading a line from a stream, or `getdelim` for reading data until a specific delimiter character is reached.

Listing 3.5 - Reading data from a stream

```

var
    SmallStr: array[0..10] of char;
begin
    //Read a block of 5 characters from the stream at a time
    //This does not test for the End of File
    fread(@SmallStr, 5, 1, StreamFilePointer);
end;

```

Listing 3.6 - Writing data to a stream

```

var
    SmallStr: array[0..4] of char;
begin
    //Writes a block of 5 characters to the stream at a time
    SmallStr := '12345';
    fwrite(@SmallStr, 5, 1, StreamFilePointer);
end;

```

Standard Input, Standard Output, and Standard Error

To obtain a file descriptor, you would normally call the `open` function with the name of the file that you want to open. On Linux, however, there are three special file descriptors that never need to be opened. These file descriptors — standard input, standard output, and standard error — are already open. Standard input is the default input mechanism, which is typically the keyboard. Standard output is the default output mechanism, which is normally the terminal where the text is displayed. Standard error is the default device to send error messages to, which is normally the same terminal that standard output is writing to.

Within the LibC.pas unit, there are three constants that represent these file descriptors for standard input, output, and error — `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, respectively.

Listing 3.7 - Using the standard file descriptors

```
Const
  { Standard File Descriptors }
  STDIN_FILENO = 0;
  STDOUT_FILENO = 1;
  STDERR_FILENO = 2;
```

When you run a console application, you will most likely see the text sent to the terminal as a result of writing to standard error, but if you are running an X Windows application it is unlikely that you will see the error messages unless you launched the X Windows application from a console window.

Listing 3.8 - Writing to standard error

```
program WritingToStandardError;

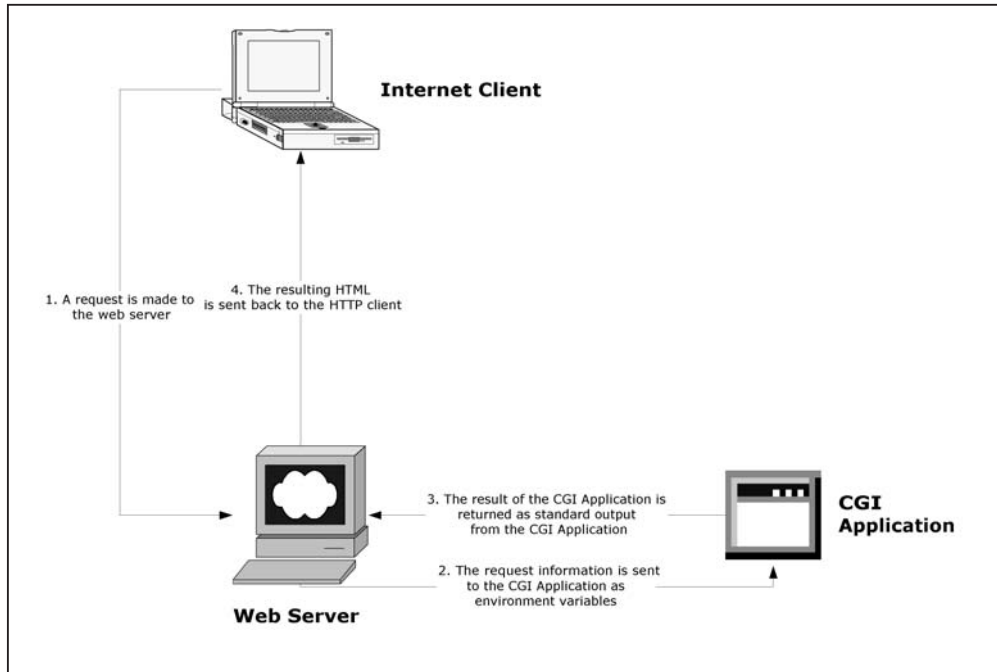
uses LibC;

var
  SimpleMessage: array[0..200] of char;

begin
  SimpleMessage := 'This message is being written'+
    ' to Standard Error';
  _write(STDERR_FILENO, SimpleMessage, 200);
end.
```

There are many applications that take advantage of using the standard input and output from within the application. Web servers will use standard input and output to process CGI Web applications. The Web server will send the information to the CGI application on standard input and will capture the returning HTML on standard output and then send the information from the standard output back to the Web server as shown in Figure 3.3.

Figure 3.3 -
How Web
servers use
standard input
and output



Converting File Descriptors to File Streams and Vice Versa

A more likely scenario for conversion is to convert a file descriptor into a valid stream. Converting between the two is a matter of calling the appropriate function. The Linux API provides the developer with several functions. These are the `fileno` function, for retrieving a file descriptor from a file stream, and the `fdopen` function, for retrieving a valid stream. Listing 3.9 demonstrates obtaining a file descriptor from a file stream.

Listing 3.9 - Obtaining a file descriptor from a file stream

```

procedure TForm1.btnBrowseClick(Sender: TObject);
var
  fileStr: PIOFile;
  filesdes: integer;
  TextContent: packed array[0..50] of char;
begin
  if OpenFileDialog1.Execute then
  begin
    edFilename.Text := OpenFileDialog1.FileName;
    //Do a simple operation on the file
    FileStr := fopen(PChar(OpenFileDialog1.FileName), 'r');

    //Create the new file descriptor
    filesdes := fileno(fileStr);

    //Read using the file descriptor
    __read(filesdes, TextContent, sizeof(TextContent));
    Memo1.Lines.Add('Read in data: ' + StrPas(TextContent) + '');
  end;
end;
  
```

```

    //Close the file descriptor
    _close(filedes);

    //Close the Stream
    fclose(FileStr);
end;
end;

```

Transferring Handles between Kylix and the API Reference

On occasion you may need to perform some operations on a file opened with Pascal's reset, write, or append functions and also need to perform some Linux file functions. Or it may be that you want to perform some operation on the file that is not available using Pascal's normal file handling syntax.

Listing 3.10 - Using the TFileRec structure

```

type
  TFileRec = packed record
    Handle: Integer;
    Mode: Word;
    Flags: Word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of Char; );
    end;

```

Whenever a file is opened using reset or write from within your application, the Kylix CLX is actually opening a file descriptor in a record structure called TFileRec. TFileRec is used internally with Kylix and is never really used outside. In Kylix the Handle field of the TFileRec record comes from a call to the open function. As a result, obtaining the file descriptor is a matter of getting the Handle field. Normal typecasting will not allow you to convert them directly. Instead, you should typecast the address of the file type to obtain the handle to the TFileRec as shown in Listing 3.11.

Listing 3.11 - Obtaining a file descriptor from a Pascal text file

```

procedure TForm1.OpenASimpleFile(strFilename: string);
var
  f: textFile;
  s: string;
  p: Pointer;
  filedes: integer;
  a: array[0..20] of char;
begin

```

```

//Open the file
AssignFile(f, strFilename);
reset(f);

//Get the Linux handle of the file descriptor
p := @f;
filedes := TFileRec(p^).Handle;

//Read a character using the Linux API.
__read(filedes, a, 20);
Memo1.Lines.Add('Read using Linux API methods. ');
Memo1.Lines.Add('' + Copy(StrPas(a), 1, 20) + '');

//Read to the file using conventional methods
readln(f, s);
Memo1.Lines.Add('Read using Pascal means: ');
Memo1.Lines.Add('' + s + '');

//Close the file
CloseFile(f);
end;

```

Converting a File Descriptor into a Stream

While you can obtain a file descriptor from a Pascal file type, you are more likely to want to work with the familiar TStream class when working with file descriptors. One of the best classes to use when working with file descriptors under Linux is the THandleStream class. The THandleStream class is a TStream wrapper around a file descriptor. In this way you can read, write, and reposition the read/write position just like any other TStream descendant. Listing 3.12 demonstrates using the THandleStream class to write some data into a temporary file.

Listing 3.12 - Using the THandleStream class

```

unit unitTestForm;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs, Libc,
  QStdCtrls;

type
  TfrmHandleStreamDemo = class(TForm)
    mmContent: TMemo;
    lblTempFilename: TLabel;
    btnOpenAndWrite: TButton;
    mmViewContent: TMemo;
    btnLoadContent: TButton;
    procedure btnOpenAndWriteClick(Sender: TObject);
    procedure btnLoadContentClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  public
    tempFileDescriptor: integer;

```

```

        TempStream: TStream;
        procedure CloseStream;
        end;

var
    frmHandleStreamDemo: TfrmHandleStreamDemo;

implementation

{$R *.xfm}

procedure TfrmHandleStreamDemo.CloseStream;
begin
    if TempStream <> nil then
    begin
        TempStream.Free;
        TempStream := nil;
        __close(tempFileDescriptor);
        tempFileDescriptor := 0;
    end;
end;

procedure TfrmHandleStreamDemo.btnOpenAndWriteClick(Sender: TObject);
var
    filename: array[0..200] of char;
begin
    if TempStream <> nil then
        CloseStream;

    filename := 'tomeskylxXXXXXX';
    tempFileDescriptor := mkstemp(@filename);
    if tempFileDescriptor <> -1 then
    begin
        TempStream := THandleStream.Create(tempFileDescriptor);
        mmContent.Lines.SaveToStream(TempStream);
        lblTempFilename.Caption := filename;
    end;
end;

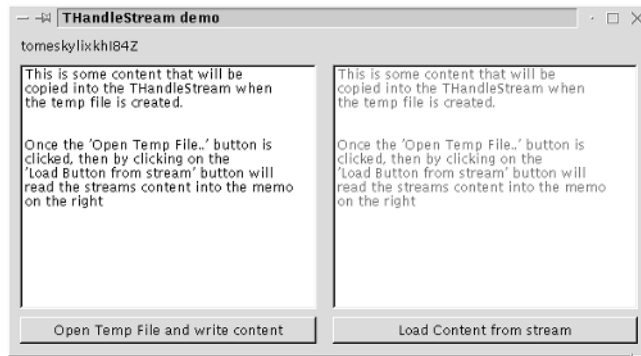
procedure TfrmHandleStreamDemo.btnLoadContentClick(Sender: TObject);
begin
    if TempStream <> nil then
    begin
        TempStream.Seek(0, soFromBeginning);
        mmViewContent.Lines.LoadFromStream(TempStream);
    end;
end;

procedure TfrmHandleStreamDemo.FormDestroy(Sender: TObject);
begin
    CloseStream;
end;

end.

```

Figure 3.4 -
Using the
THandle-
Stream class



Because the TFileStream class derives from THandleStream, you can obtain the file descriptor of the TFileStream by retrieving the value of the Handle property.

File Permissions and Ownership

Linux is a multi-user operating system, and as such, it needs to have permission to protect files owned by one user from other users. Linux does this by assigning permissions to each file, so that only the users with the appropriate permissions can do particular operations on a file.

As a result, within a multi-user operating system such as Linux, all files and directories are owned by a particular user, with the owner of a file typically being the user who originally created the file or directory. In many cases, ownership of a file or directory can be taken using a Linux command such as `chown`.

A file or directory also has a particular user group. The first example below demonstrates changing the owner of a group, while the second demonstrates changing the owner and the group of the file.

```
chown glenn WebImage.gif
chown glenn:friends index.html
```

Setting permissions on files and directories under Linux is normally done using the `chmod` utility. The effects of any permission changes can be seen when a directory listing is normally attained from an `ls` command as shown in Figure 3.6. You can also look at the permissions of the file using an application such as KFM, the file manager application for KDE, as shown in Figure 3.5.

Ownership of files and group ownership is important because this is how the permissions in Linux are managed. There are separate permissions for the owner of the file, separate permissions for users within the file's ownership group, and then permissions for everyone else. This is the way of Linux security and knowing this will assist in developing and debugging multi-user applications.

Figure 3.5 - Viewing permissions and ownership of a file

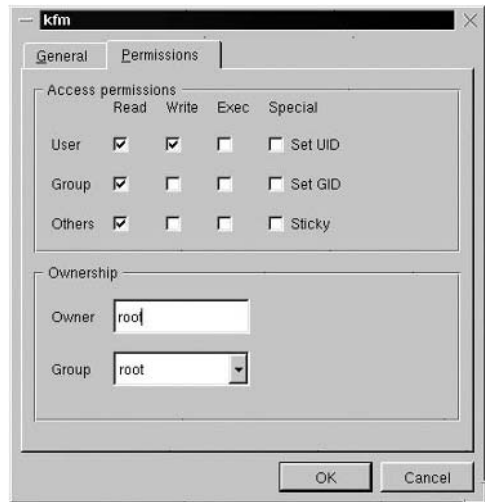
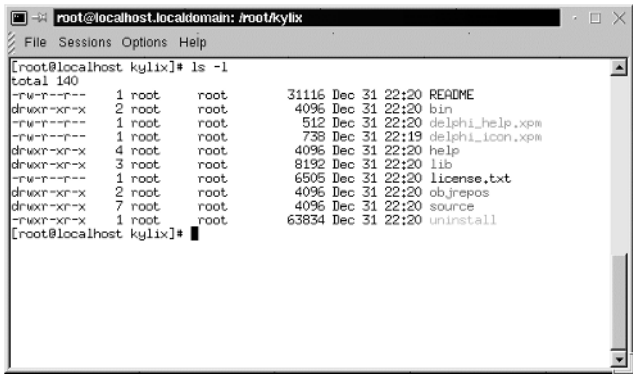


Figure 3.6 - Looking at a file and directory permissions using the `ls` command



If you examine Figure 3.6 for any given file or directory, there is a list of characters that represent the details about the file or directory and look like the character string “drwxr-xr-x”.

The first character denotes if the listing of the entry is a file, directory, or linked file — “d” if it is a directory, “l” for a linked file, or blank for a file. Each file then has three sets of permissions: one set of permissions for the owner of the file, one set of permissions for the group that the file is in, and another set of permissions for everyone else.

The “rwx” block represents reading, writing, and execution rights on the file or directory. Setting read (r) on a file allows the file to be read. Setting read on a directory allows the directory contents to be listed.

Setting write (w) on a file allows the file to be modified or deleted. Setting the write modifier on a directory gives the user the ability to modify files in that directory or to create new files within that directory.

Setting execute (x) on a file allows the file to be run as an application.

Adding execute permissions for the owner of the file using the `chmod` command would be done as follows:

```
chmod u+x <filename>
```

Removing write access for other users in the file's group would be:

```
chmod g-w <filename>
```

Understanding permissions is extremely important when developing applications for Linux, not only for setting the appropriate permissions, but also for debugging applications. For instance, your application might not have correct permissions to access a file, or it may be that a user is being impersonated in an application such as a Web server, and as a result, your application does not have enough permissions to complete the desired operation.

When developing applications with Kylix, if you were using standard Pascal file data types such as `File` or `TextFile` when creating the file, you do not have the ability to perform operations that specify what file permissions the newly created file will have. Instead, when using a file descriptor, you can specify the security options in the call to the open function.

In a similar fashion, when creating directories with Kylix, you would normally use the `MkDir` function found in the `System` unit. This function will only create a directory with default permissions for the new directory. Using the `__mkdir` function found in the `LibC` unit, however, will allow you to specify the permissions for that directory when it is created.

Likewise, the `creat` function will allow you to specify the permissions of a file when the file is created. In other file functions that are used within the Linux API, you will see that the permissions can often be specified when a file or directory is created.

```
function __mkdir(PathName: PChar; Mode: Integer): Integer; cdecl;
function creat(FileName: PChar; Mode: Integer): Integer; cdecl;
```

Table 3.2 - Possible permission mode values

Permission Constant	chmod Command Setting	Description
<code>S_IRUSR, S_IREAD</code>	<code>u+r</code>	On a file, this will give the owner of the file the ability to read the file. On a directory, this gives the owner of the directory the ability to list the contents of the directory.
<code>S_IWUSR, S_IWRITE</code>	<code>u+w</code>	On a file, this allows the user to modify the file. On a directory, this gives the ability to create new files within that directory.
<code>S_IXUSR, S_IEXEC</code>	<code>u+x</code>	This setting applies only to files and allows the file to be run as an executable. For example, Kylix sets the executable permission on an application whenever you compile the application. The new application is generated as a file, and then permission is set to allow the new application to be executed.
<code>S_IRWXU</code>	<code>u+rwX</code>	This combines the functionality of <code>S_IRUSR</code> , <code>S_IWUSR</code> , and <code>S_IXUSR</code> . Its value is equivalent to <code>S_IRUSR</code> , <code>S_IWUSR</code> , or <code>S_IXUSR</code> .

Permission Constant	chmod Command Setting	Description
S_IRGRP	g+r	On a file, this permission allows users in the files group to have read access to the file. On a directory, this allows the users in the files groups to list the contents of the directory.
S_IWGRP	g+w	On a file, this gives users in the files group the ability to modify the file.
S_IXGRP	g+x	This permission allows users in the files group to execute the file.
S_IRWXG	g+rxw	This constant combines S_IRGRP, S_IWGRP, and S_IXGRP.
S_IROTH	o+r	Allows all other users read access to the file or directory.
S_IWOTH	o+w	Allows all other users write access to the file or directory.
S_IXOTH	o+x	Allows all other users execute access to the file.
S_IRWXO	o+rxw	This value is the combination of S_IROTH, S_IWOTH, and S_IXOTH.
S_ISUID		This permission will set the user ID on execution of the application. Its main use is to allow users other than the administrator to execute the application with administrator-level privileges. For more information, see Chapter 4.
S_ISGID		This permission will set the group ID on execution of the application. Like the S_ISUID contact, its use is to allow users other than the administrator to execute the application with administrator-level privileges. For more information, see Chapter 4.
S_ISVTX		This permission is set on directories and allows users to delete the file only if they are the owner of that file.

The mode parameter will create the permissions for a directory or a file based on a list of constants found in the LibC unit that correspond to similar values when using the Linux chmod command. The corresponding permissions and their mode constant can be seen in Table 3.2.

As you can see, assigning permissions for a particular file or directory is a matter of combining the constants of Table 3.2 to allocate the user's appropriate levels of access, such as:

```
__mkdir('/usr/glenn/newdir', S_IRWXU or S_IRGRP);
```


Getting Information and Attributes about a File

Listing 3.18 demonstrates that you can determine information about a particular entry in a directory stream by using the `stat` function. In fact, the `stat` group of functions, which includes `stat`, `lstat`, `fstat`, `stat64`, `lstat64`, and `fstat64`, can all access information on a particular file or directory entry.

The `stat` group of functions returns a data type `TStatBuf` as shown in Listing 3.13.

Listing 3.13 - Using the `TStatBuf` data type

```
type
  TStatBuf = record
    st_dev: __dev_t;           { Device. }
    __pad1: Word;
    st_ino: __ino_t;          { File serial number. }
    st_mode: __mode_t;        { File mode. }
    st_nlink: __nlink_t;      { Link count. }
    st_uid: __uid_t;          { User ID of the file's owner. }
    st_gid: __gid_t;          { Group ID of the file's group. }
    st_rdev: __dev_t;         { Device number, if device. }
    __pad2: Word;
    st_size: __off_t;         { Size of file, in bytes. }
    st_blksize: LongWord;     { Optimal block size for I/O. }
    st_blocks: __blkcnt_t;    { Number 512-byte blocks allocated. }
    st_atime: __time_t;       { Time of last access. }
    __unused1: LongWord;
    st_mtime: __time_t;       { Time of last modification. }
    __unused2: LongWord;
    st_ctime: __time_t;       { Time of last status change. }
    __unused3: LongWord;
    __unused4: LongWord;
    __unused5: LongWord;
  end;
```

The `st_dev` field represents the device ID of the file. The `st_ino` field refers to the file's serial number for the device that it is located on.

`st_mode` represents both the permissions on the file and also what type of file it is. To check permissions of the file, you can simply test against the appropriate permission constant from Table 3.2. Listing 3.14 demonstrates examining the permissions of a particular file.

Listing 3.14 - Displaying the permissions for a file

```
procedure TForm1.DisplayPermissions(strFilename: string);
var
  buf: TStatBuf;
begin
  stat(PChar(strFilename), buf);

  //Analyze the Permissions (Owner)
  if buf.st_mode and S_IRUSR > 0 then
    StringGrid1.Cells[1, 1] := 'Yes';
  if buf.st_mode and S_IWUSR > 0 then
    StringGrid1.Cells[1, 2] := 'Yes';
  if buf.st_mode and S_IXUSR > 0 then
```

```

StringGrid1.Cells[1, 3] := 'Yes';

//Analyze the Permissions (Group)
if buf.st_mode and S_IRGRP > 0 then
    StringGrid1.Cells[2, 1] := 'Yes';
if buf.st_mode and S_IWGRP > 0 then
    StringGrid1.Cells[2, 2] := 'Yes';
if buf.st_mode and S_IXGRP > 0 then
    StringGrid1.Cells[2, 3] := 'Yes';

//Analyze the Permissions (Others)
if buf.st_mode and S_IROTH > 0 then
    StringGrid1.Cells[3, 1] := 'Yes';
if buf.st_mode and S_IWOTH > 0 then
    StringGrid1.Cells[3, 2] := 'Yes';
if buf.st_mode and S_IXOTH > 0 then
    StringGrid1.Cells[3, 3] := 'Yes';
end;

```

The `st_mode` field also specifies the type of the file. This value can specify if a value is a file or directory, a blocked file, or a file-in/file-out (FIFO). The constants listed in Listing 3.15 allow you to test the type of file.

Listing 3.15 - Examining the type of file

```

const
    __S_IFDIR      = $4000; { Directory.  }
    __S_IFCHR      = $2000; { Character device.  }
    __S_IFBLK      = $6000; { Block device.  }
    __S_IFREG      = $8000; { Regular file.  }
    __S_IFIFO      = $1000; { FIFO.  }

    ...

//See if the file is a fifo
stat('/usr/duke/somefile', sbuf);
if sbuf.st_mode and __S_IFIFO > 0 then
begin
    //The file is a fifo
    showmessage('We have a fifo');
end;

```

The `st_nlink` field of the `TStatBuf` structure holds the information about the number of links to the file or directory that exist. The `st_uid` and `st_gid` fields hold a reference to the id of the owner's file and group.

The `st_size` field actually represents the size of the file in bytes, where `st_blksize` represents the recommended size of the block that should be used when performing block read or write operations on the file and `st_blocks` represents the amount of 512-byte blocks that have been allocated to store the file.

The last fields worth examining in the `TStatBuf` record structure are the fields that hold date and time information. The `st_atime` field contains the date and time the file was last accessed, the `st_mtime` field holds the time the file was last modified, and the `st_ctime` field holds the last time the file had a status change.

To make sense of the `__time_t` data type, simply use the `localtime` function to return a pointer to a `TUnixTime` data structure as shown in Listing 3.16.

Listing 3.16 - Using the `TUnixTime` data type

```
Type
tm = packed record
    tm_sec: Integer;           { Seconds. [0-60] (1 leap second) }
    tm_min: Integer;          { Minutes. [0-59] }
    tm_hour: Integer;          { Hours. [0-23] }
    tm_mday: Integer;          { Day. [1-31] }
    tm_mon: Integer;           { Month. [0-11] }
    tm_year: Integer;          { Year - 1900. }
    tm_wday: Integer;          { Day of week. [0-6] }
    tm_yday: Integer;          { Days in year. [0-365] }
    tm_isdst: Integer;         { DST. [-1/0/1] }
    tm_gmtoff: Integer;        { Seconds east of UTC. }
    tm_zone: ^Char;           { Timezone abbreviation. }
end;
TUnixTime = tm;
PUnixTime = ^TUnixTime;
```

Listing 3.17 - Converting the integer time

```
function IntTimeToDateTime(Value: integer): TDateTime;
var
    pt: PUnixTime;
    wHour, wMinute, wSecond, wMilliSecond,
    wDay, wMonth, wYear: word;
begin
    //Convert to a TUnixTime structure
    pt := localtime(@Value);
    with pt^ do
    begin
        //Get Time Information
        wHour := tm_hour;
        wMinute := tm_min;
        wSecond := tm_sec;
        wMilliSecond := 0;

        //Get Date information
        wDay := tm_mday;
        wMonth := 1 + tm_mon;
        wYear := 1900 + tm_year;

        Result := EncodeDate(wYear, wMonth, wDay) +
            EncodeTime(wHour, wMinute, wSecond, wMilliSecond);
    end;
end;
```

Scanning Files in a Directory

Linux has a fantastic way of reading the entries in a directory, and the same techniques you use to open a file are used to open a directory. You open the directory using the `opendir` function, read the entries from the directory using the `readdir` function, and once you have completed your actions on the directory, you close the directory using the `closedir` function. Listing 3.18 demonstrates a `TListBox` descendant that reads from a directory and populates the `TListBox` `Items` property with the names of the directories and files.

Listing 3.18 - Reading a list of files in a directory

```
procedure TTKFileListBox.LoadDirectoryEntries;
var
  directoryHandle: PDirectoryStream;
  PDirInfo: PDirEnt;
  sbuf: TStatBuf;
begin
  Items.Clear;
  //Here is where we read the entries of the directory
  directoryHandle := opendir(PChar(FDirectory));

  if directoryHandle <> nil then
    try
      PDirInfo := readdir(directoryHandle);
      while PDirInfo <> nil do
        begin
          //In reality the PDirInfo^.d_type should hold the information
          //if the file is a directory, but this is not implemented
          //in all versions. The stat function is more reliable.

          //Get information about a file
          stat(PChar(FDirectory + '/' + PDirInfo^.d_name), sbuf);
          if sbuf.st_mode and __S_IFDIR > 0 then
            begin
              //This is a directory
              Items.Add([''+StrPas(PDirInfo^.d_name)+'']);
            end else begin
              //This is a file
              Items.Add(PDirInfo^.d_name);
            end;

            PDirInfo := readdir(directoryHandle);
          end;
        finally
          closedir(directoryHandle);
        end else begin
          raise ETKFileListBoxError.Create('The directory '''+FDirectory+
            ''' could not be opened.');
```

Because of this likeness to using files, when working with directories you will often hear them referred to as directory streams.

Just as with a file descriptor or stream, you can go to specific points in the file using the `seek` and `tell` functions and you can navigate within positions of a directory stream using the `seekdir` and `telldir` function calls.

Working with Buffers

When you write a character or a small block of data to file, you would be excused for thinking that the data is written directly to disk. Unfortunately, under Linux, writing data to a file stream is usually buffered in memory before it is written to disk. Writing to silicon will always be faster than writing to a hard drive. By writing to memory first and waiting until it has enough to make an efficient write to the hard disk, the performance of your application will benefit.

But it may be the case that you need to ensure that the information you write to disk writes to the disk immediately. Using the Linux API, you have the ability to choose between three different buffering techniques to specify the way the information is written to the disk.

The first method, and the one used by default when you are writing to a file, is full buffering. With full buffering, data is written to memory, and when it is effective to write the data, it will place the information to disk.

The second method is line buffering, which writes content to the disk whenever data written to the file reaches content with an end-of-line character included. At this point the data stored in the buffer's memory, including the end-of-line character, is written to the file. In this way data is written to the file line by line.

The third method is unbuffered. In this mode, data is immediately written to disk.

Specifying which mode you use is a matter of using `setvbuf` to specify the buffering mode. You should note that even though buffers might not have written content to disk, buffered data will be written to disk when the `fflush` function is called on a stream, when the stream is closed, or when the application has closed.

File Operations (Renaming, Copying)

If you have been using UNIX or Linux for any period of time, you have undoubtedly come across the basic functions that are used to copy, rename, or delete files, and all manner of operations on files and directories.

If there is a file or directory operation that you would like to perform, it is quite likely that the function call has the same name as the Linux command or is an expanded version of the Linux command name. Examples of these include `rename`, `remove`, `__chdir`, and `__mkdir`, among others.

Many of the functions to do basic operations have been renamed, such as the `__chdir` function, to avoid confusion with the `chdir` function found in the System unit.

A handy little trick that you can use if the Linux command that you want to perform is not available through an API interface is to use the `__system` function to call that function with the parameters and send the output to a file.

Listing 3.19 demonstrates using the `__system` function to find the word count for a particular file using the `wc` function.

```
[root@localhost kylix]# wc -w README
4424 README
```

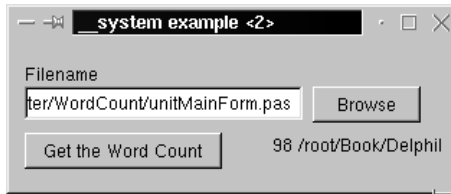


Figure 3.7 - Using the LibC `__system` function to calculate the word count of a file

Listing 3.19 - Obtaining the word count of a text file

```
procedure TForm1.Button1Click(Sender: TObject);
var
  sl: TStringList;
begin
  //Execute the Command
  LibC.system(PChar('wc -w '+edFilename.Text+' > /tmp/filewc.txt'));
  //Load the content of the operation.
  sl := TStringList.Create;
  try
    sl.LoadFromFile('/tmp/filewc.txt');
    //Display the details.
    lblWordCount.Caption := sl.Text;
  finally
    sl.Free;
  end;
end;
```

The ioctl Function

As we have seen before, most devices on Linux are treated as files, such as sockets, scanners, and other devices. Although most devices, such as files, sockets, and scanners, work in the same way (by reading and writing data), all of these devices can use the file descriptor's standard way of reading and writing data using the `_read` and `_write` functions. Many devices have other operations that they can perform, such as when a CD-ROM device ejects the disk or plays an audio track. The `ioctl` function is used to extend the ability of these generic devices so that they can perform operations specific to the device.

```
function ioctl(__fd: Integer; __request: LongWord): Integer; cdecl; varargs;
```

The `ioctl` function essentially takes an open file descriptor — the command that is to be executed. Then, if the command requires additional parameters, `varargs` allows additional arguments to be passed into the function. While the `ioctl` function is part of Kylix's implementation of the Linux API, many of the commands that are used in the call to `ioctl` are not found in `LibC.pas`. As a result, when using this function in conjunction with a device, you may have to convert the commands from C into their Object Pascal equivalents using the techniques discussed in Chapter 8.

Listing 3.20 - Ejecting a CD-ROM device

```

procedure EjectCDROM;
var
  cdDrive: integer;
const
  CDROMEJECT = $5309; //Converted from the C header file cdrom.h
begin
  //First we open the device to get the cdrom
  cdDrive := open('/dev/cdrom', O_RDONLY or O_NONBLOCK);

  //Let's now eject the CD with the ioctl function
  ioctl(cdDrive, CDROMEJECT);

  //Let's close the device
  _close(cdDrive);
end;

```

Listing 3.20 demonstrates using the `ioctl` function to send a command directly to the CD-ROM to eject the CD. The constants used in this function are converted from the C header file `cdrom.h` as the constant `CDROMEJECT` is not declared in `LibC.pas`.

You can use the `ioctl` function on many occasions when you find that you need finer control over a device or to perform a task that you cannot do with normal file operations.

Creating Links (Shortcuts for Linux)

If you have worked with Windows 95 and above for any period of time, you should be familiar with the concept of shortcuts. A shortcut is a pseudo-directory or file that actually opens up a particular destination file or directory. Under Linux this functionality is called links and has actually been around much longer than the Windows implementation of shortcuts.

Under Linux, it is quite easy to create a link file using the `ln` command by passing in the source file to link.

```
ln Custinfo.dat customerInfo.data
```

When you open a link file, the Linux system will actually open the destination of the link file. You should be aware that you can have a link that actually links to another file. Many Linux API calls will try to resolve links to link files and eventually get to an actual file. If the system cannot resolve the link to an actual file or directory, the function that accesses the files through links may fail and the value `ELOOP` will be returned by a call to the `errno` function.

File Conventions

When looking through the API listing you will see that there are several conventions used for function names on Linux. There are many functions that perform a particular task and other functions that do the same thing but have a slightly different function name. The functions that end in “64” are a prime example.

The functions that end in “64” are used on Linux systems that are not 64-bit compliant, and they allow these non 64-bit compliant machines to work with files over $2^{32}-1$ bytes in

length. If you examine the functions `open` and `open64`, you will see that they do essentially the same thing. However the `open64` function allows the developer to access files larger than $2^{32}-1$ bytes in size on systems that are not 64-bit compliant. On all of these occasions where a function and its 64-bit equivalent are found, the functions will use different data types as file handle IDs.

Unlocked Functions

The Linux API functions originally dealt with file input in a serial manner, and the functions expected that there would only be serial access to a file within an application, due to the fact that only one thread of execution would be running.

As threads were introduced to Linux functions, applications would need safe and reliable access to files.

Functions ending with “_unlocked” are the extensions to the normal file functions that operate on threadsafe applications. The _unlocked functions are first opened as a normal file stream with the `fopen` function. Each thread that wants to access the file then locks the file with the `flockfile` function that keeps a counter of which threads have access to the file.

Although the normal functions for reading and writing are threadsafe, locking stream using `flockfile` will allow you to perform a sequence of operations on a stream and guarantee that the thread that is performing the operation has dedicated access to the stream.

varargs — A New Declaration for Kylix

As the LibC unit is based on the C library, it stands to reason that many of the C-style idiosyncrasies would find their way into Kylix. If you have ever coded in C, you would no doubt have come across the function `printf`.

Listing 3.21 - Using the `printf` function in a C application

```
/* Simple C code */
printf("Some Number is %d and some string is %s", 10, "Hi Everyone");
```

In this function you create a string which defines the format of the string that you wish to write and a collection of value tokens that are defined by the percent (%) symbol followed by characters that define the data type and presentation of the value that this token would replace. There are several types of delimiters that are often used with the `printf` function. The Kylix declaration of the `printf` function is shown in Listing 3.22.

Listing 3.22 - Using the `printf` function when in Kylix

```
function printf(Format: PChar): Integer; cdecl; varargs;

//Calling the printf function from Kylix
printf('This is a number %d; and this is a string %s', 10, 'Hello World');
```

As you can see, the only parameter in the Kylix declaration of the function is the `Format` that defines the first string. The `varargs` parameter is a list of arguments of any data type that is passed as the last group of parameters. In Listing 3.22, the actual var arguments are the number 10 and the string “Hello World.”

Throughout this book you will see many references to functions that have varargs directives. The varargs directive is a very powerful feature that allows a function to be extendable by allowing alternative values in the parameter list.



Note: varargs is very similar to the array of const declaration that is used in many of CLX's methods, in which a list of values of any type can be passed in. The only real difference is that the varargs directive does not require the brackets around the list of values. In fact, it may be better to stick with the format function found in the SysUtils unit. It has all the same functionality as printf, but works better with the standard Object Pascal data types.

File Errors

As you discovered in Chapter 2, the majority of functions use the Linux API standard of returning a result of 0 when the function is successful and returning -1 when the function is unsuccessful.

When a function call is unsuccessful, you can check the reason why by using the techniques presented in Chapter 2. The normal method to get the error code is by calling the `errno` function. When working with files you are more likely to come across errors from the list in Table 3.3.

Table 3.3 - Common Linux API errors when working with files

Getting this error	Means that...
EACCES	You do not have the appropriate user permission to search in this directory.
ENAMETOOLONG	The filename that you are trying to create or use in a rename function is larger than the allowed length.
ENOENT	You have tried to open a directory or file that does not exist.
ENOTDIR	You have tried to open a directory, but it was a file rather than a directory.
ELOOP	When symbolic links are created, too many are being resolved to try and access the file or directory.
EEXIST	The file already exists.
EFBIG	The file is too large.
EIO	There was an input/output error while trying to perform the function.
EMFILE	There are too many open files.
ENFILE	There are too many open files in your process.
ENOTEMPTY	The directory is not empty. This is usually a result of trying to remove a directory.
ENOMEM	Not enough space to perform this function.

API Reference

__chdir *LibC.pas*

Syntax

```
function __chdir(
  PathName: PChar;
):Integer;
```

Description

This function changes the working directory within your application.

Parameters

PathName: This parameter holds the name of the directory that you wish to change to. You must ensure that PathName is an actual directory.

Return Value

The function returns 0 if successful and -1 if unsuccessful. If the function is unsuccessful, calling `errno` will return `EFAULT`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`, `EACCES`, `ELOOP`, or `EIO`.

See Also

`getcwd`

Example

Listing 3.23 - Using the __chdir function

```
__chdir('.'); //Change to the parent directory
__chdir('/root/'); //Change to the fixed path '/root'
__chdir('images'); //Change to the relative path 'images'
```

__close *LibC.pas*

Syntax

```
function __close(
  Handle: Integer;
):Integer;
```

Description

This function will close an open file descriptor created by the `open` function. You should not confuse this with the `System.pas` `close` function.

Parameters

Handle: This is the file descriptor that is used to close the file. If the value passed into this parameter is not a valid file descriptor, the function will fail and the `EBADF` value will be returned from `errno`.

Return Value

The function will return 0 if successful and -1 if unsuccessful.

See Also

open

*Example***Listing 3.24 - Using the `__close` function**

```
Var
    FileHandle: integer;
begin
    FileHandle := open(PChar(edFilename.Text), 0_RDWR, 0);
    //Do some operations on the file
    __close(FileHandle);
end;
```

`__mkdir` LibC.pas*Syntax*

```
function __mkdir(
    PathName: PChar;
    Mode: Integer
):Integer;
```

Description

This function will create a new blank directory in the directory denoted in PathName.

Parameters

PathName: The name of the physical directory that is to be created.

Mode: This sets the permissions for the directory that is created. This value is OR'ed with a group of the access options that are available and will consist of a combination of values from Table 3.2.

Return Value

The `__mkdir` function returns 0 when successful. If the directory could not be created, it will return -1. A call to `errno` will display the reason that the function was unsuccessful.

See Also

`__chdir`

*Example***Listing 3.25 - Using the `__mkdir` function**

```
procedure TForm1.MakeSomeDirectories;
begin
    __mkdir('/tmp/MyNewDirectory', S_IRUSR or S_IWUSR or
```

```

        S_IRGRP or S_IROTH or S_IWOTH);
end;

```

__read LibC.pas

Syntax

```

function __read(
  Handle: Integer;
  var Buffer;
  Count: Integer
):Integer;

```

Description

The `__read` function will copy data from the file descriptor denoted by `Handle` at the current file descriptor's read/write position. Data is read into from the file descriptor in the memory location at `Buffer` and the size of the data is denoted by the `Count` parameter. This function can be used to read from any file descriptor including pipes and FIFOs. Within this function the underscore symbols are used to avoid calling the `System.read` function.

Parameters

`Handle`: This is the handle of an open file descriptor.

`Buffer`: This is the location of the data that is to be read in. This is a pointer to some memory buffer or variable address.

`Count`: This is the size in bytes of how much data to read in. For this parameter it is best to use the `sizeof` function and pass in the data type that you want to read in.

Return Value

The function returns 0 if successful and `-1` when unsuccessful. When unsuccessful, calling the `errno` function will return the error that occurred.

See Also

`fread`

Example

Listing 3.26 - Using the `__read` function

```

type
  TCustomer = record
    CustName: string[40];
    CustEmail: string[50];
    Age: integer;
    SpecialMember: boolean;
  end;

procedure TForm1.ReadInACustomer;
var
  c: TCustomer;
  intFileDes: integer;

```

```

begin
    intFileDes := open('CustomerInfo', 0_RDONLY, 0);
    __read(intFileDes, c, sizeof(TCustomer));
    __close(intFileDes);
end;

```

__rename LibC.pas

Syntax

```

function __rename(
    OldName: PChar;
    NewName: PChar
):Integer;

```

Description

This function will change the filename from one name to another name. This function will work on files and directories. Using this function is similar to copying the file to the destination filename and then removing the old file; however, using the rename function will simply update the file allocation table internally, which is more efficient.

Parameters

OldName: This is the name of the file or directory that will be renamed.

NewName: This parameter holds the filename or directory that the file will be called. If it is a directory, the directory should not exist or should be empty.

Return Value

When successful, this function will return 0. If the function fails, it returns -1. The function may fail as a result of an invalid filename, because of incorrect permissions on the file that is to be renamed (errno will return EACCES), or because the directory or file is already in use (errno will return EBUSY).

If the function fails and you are copying a directory, ENOTEMPTY will be returned by the errno function if the destination directory is not a blank directory. If you try to rename a directory to a subdirectory of itself, you will get the EINVAL errno value.

If the source filename or directory does not exist on the file system, a value of ENOENT will be returned from the errno function.

If the source and the destination files are on two different file systems, then errno will return EXDEV. In this case it would be better to copy the file from the source to the destination and then delete the source file.

Example

Listing 3.27 - Using the __rename function

```

//Rename a file
__rename('CustomerInfo', 'Custinfo.data');

```

__truncate LibC.pas**Syntax**

```
function __truncate(
  const FileName: PChar;
  Length: __off_t
): Integer;
```

Description

The truncate function will alter the size of a file to a particular file size of Length bytes.

Parameters

FileName: This is the file that is to have its size altered.

Length: This will become the new size of the file. Normally, when truncating a file you have the option to shrink the file to the new file size, but you can also specify that the file size actually grow by specifying a Length parameter larger than the existing size of the file.

Return Value

On success this function returns 0. If the function fails, it returns -1.

See Also

fopen, stat

Example**Listing 3.28 - Using the __truncate function**

```
//Let's shrink the file
__truncate(PChar(edFilename.Text), 50);
```

__write LibC.pas**Syntax**

```
function __write(
  Handle: Integer;
  const Buffer: Integer;
  Count: Integer
): Integer;
```

Description

The __write function places data into the open file descriptor Handle. Data will be copied from the Buffer parameter to the file. The amount of data that is written to the file in bytes is specified in the Count parameter. Additionally, this function can be used by pipes or FIFOs.

Parameters

Handle: This is the file descriptor handle that is used. This value is typically a value returned by the open function or the pipe function.

Buffer: This holds the location of the data that will be copied from memory to the file. This is often simply a pointer to a variable that will be written out.

Count: This is the amount of data that will be transferred to the file in bytes. The sizeof function should be used to safely calculate the amount of data to transfer.

Return Value

This function will return 0 if successful and -1 if unsuccessful.

See Also

__read, open, __close

Example

Listing 3.29 - Using the __write function

```
type
  TCustomer = record
    CustName: string[40];
    CustEmail: string[50];
    Age: integer;
    SpecialMember: boolean;
  end;

procedure TForm1.WriteACustomer;
var
  c: TCustomer;
  intFileDes: integer;
begin
  intFileDes := open('CustomerInfo', 0_WRONLY, 0);
  c.CustName := 'Glenn Stephens';
  c.CustEmail := 'glenn@coderaage.com';
  c.Age := 28;
  c.SpecialMember := true;
  __write(intFileDes, c, sizeof(TCustomer));
  __close(intFileDes);
end;
```

alphasort, alphasort64 *LibC.pas*

Syntax

```
function alphasort(
  const e1: Pointer;
  const e2: Pointer
):Integer;
```

```
function alphasort64(
  const e1: Pointer;
```

```
const e2: Pointer
):Integer;
```

Description

The `alphasort` function is a built-in sorting routine designed to arrange a list of directory entries in alphabetical order. This function is normally passed to another function, such as `scandir`, so that it can be used when arranging the entries of a directory.

Parameters

`e1`: This is a pointer to a `TDirEnt` record used for comparing directories.

`e2`: This is also a pointer to a `TDirEnt` record used for comparing directories.

Return Value

The function returns the value of a string comparison which calls the `strcoll` function.

See Also

`scandir`, `versionsort`, `versionsort64`

Example

Listing 3.30 - Using the `alphasort` function

```
var
  dirCount: integer;
  entries: PPDirent;
begin
  //Use the alphasort
  dirCount := scandir(PChar(edDirectory.Text), entries, nil, alphasort);
end;
```

chmod *LibC.pas*

Syntax

```
function chmod(
  FileName: PChar;
  Mode: LongWord
):Integer;
```

Description

This function changes permissions on a file or directory to grant or deny specific access to the owner of the file/directory, the file or directory group of the file descriptor, and other users.

Parameters

`FileName`: This is the name of the file or directory that will have its permission level changed.

`Mode`: This is the collection of permissions that will be set. The values come from Table 3.2 and represent which permissions are to be set.

If you are adding or removing permissions to a file, it is best to retrieve the current permissions by using the `stat` function and then incorporate the changes into the permissions as an addition.

Return Value

When the function is successful, it returns 0. If the function fails, it will return `-1`. A failed function is usually due to an invalid filename parameter or because the file is located on a directory that is on a read-only system such as a CD-ROM.

See Also

`fchmod`, `stat`

Example

Listing 3.31 - Using the `chmod` function

```
//Allow the owner of the file full access
//and the file group read permission
chmod('/usr/glenn/readme.txt', S_IRWXU or S_IRGRP);
```

clearerr, clearerr_unlocked **LibC.pas**

Syntax

```
procedure clearerr(
  Stream: PIOFile
);

procedure clearerr_unlocked(
  Stream: PIOFile
);
```

Description

This function clears a stream's error flag and end of file flag. The end of file and error flags are normally set by reading data from a file until the end of file has been reached.

Parameters

Stream: This is the name of the stream that will have its end of file and error flags cleared.

See Also

`ferror`, `feof`

Example

Listing 3.32 - Using the `clearerr` function

```
procedure TForm1.Button3Click(Sender: TObject);
var
  SourceStream: PIOFile;
  DestStream: PIOFile;
  c: integer;
```

```

begin
    //Copy the file by copying individual bytes.

    //Open the Source file
    SourceStream := fopen(PChar(edSourceFile.Text), 'r');

    //Open the destination file (create if needed)
    DestStream := fopen(PChar(edDestFile.Text), 'w+');

    //Copy the individual bytes
    while (feof(SourceStream) = 0) do
        begin
            c := getc(SourceStream);
            putc(c, DestStream);
        end;

    //Clear the EOF and Error markers of the file
    clearerr(SourceStream);

    //Close the streams
    fclose(DestStream);
    fclose(SourceStream);
end;

```

closedir ***LibC.pas***

Syntax

```

function closedir(
    Handle: PDirectoryStream
):Integer;

```

Description

This function closes a directory stream after it has been opened with the `opendir` function.

Parameters

Handle: This is the directory stream handle value that is returned from a call to the `opendir` function.

Return Value

This function returns 0 when successful and -1 when the function fails. The most likely reason that the function would fail is due to the `Handle` parameter being invalid. In this case, the `errno` function will return the error `EBADF`.

See Also

`opendir`, `readdir`, `seekdir`, `telldir`

*Example***Listing 3.33 - Using the closedir function**

```

var
  directoryHandle: PDirectoryStream;
  PDirInfo: PDirEnt;
begin
  //Here is where we read the entries of the directory
  directoryHandle := opendir(PChar(FDirectory));

  if directoryHandle <> nil then
    try
      PDirInfo := readdir(directoryHandle);
      while PDirInfo <> nil do
        begin
          Items.Add(PDirInfo^.d_name);
          PDirInfo := readdir(directoryHandle);
        end;
      finally
        closedir(directoryHandle);
      end;
    end;
  end;
end;

```

creat, creat64 LibC.pas*Syntax*

```

function creat(
  FileName: PChar;
  Mode: Integer
):Integer;

```

```

function creat64(
  FileName: PChar;
  Mode: Integer
):Integer;

```

Description

The creat function creates a new file, opens the file, and then returns a valid file descriptor. It is similar to the open function, except that the file will be created before it is open. It is equivalent to calling open with the flags O_CREAT, O_WRONLY, or O_TRUNC. The creat64 version of this function is functionally equivalent to the creat, but can be used on non-64 bit machines to work with files larger than $2^{32}-1$ bits.

Parameters

FileName: This is the filename that is to be created. Data can be written to this file upon a successful function call.

Mode: This parameter holds the permissions information on the particular file. The selected values from Table 3.2 should be OR'ed together to arrive at the desired permissions level for your file.

Return Value

When successful, this function will return a valid file descriptor. If the function fails, it will return `-1`, in which case you should call the `errno` function to see the cause of the error.

The `errno` function will return `EEXIST` if the filename already exists, `EACCES` if you did not have the correct permissions to perform the operation, `ENAMETOOLONG` if the filename that you tried to create was too large, or `EROFS` if the file you tried to create was on a read-only file system such as a CD-ROM. You may get `ELOOP` if the directory that you tried to create required too many link jumps to find the desired file. `ENOSPC` will be returned if the disk is out of space. `EMFILE` will be returned if there are already too many files open or `ENFILE` if the total amount of files allowed to be open on the file system is obtained.

See Also

`open`, `open64`

Example

Listing 3.34 - Using the `creat` function

```
procedure TForm1.Button2Click(Sender: TObject);
var
  fd: integer;
  Mesg: array[0..100] of char;
begin
  fd := creat(PChar(edFilename.Text), S_IRWXU or S_IRGRP);
  Mesg := 'This is my newly created File.'+#0;
  __write(fd, Mesg, Length(Mesg));
  __close(fd);
end;
```

dirfd *LibC.pas*

Syntax

```
function dirfd(
  Handle: PDirectoryStream
):Integer;
```

Description

This function returns a file descriptor from a currently open directory stream. Obtaining a file descriptor maybe useful if you wish to monitor the status of a directory (i.e., using the `select` function to see if any files have been written to that directory.)

Parameters

Handle: This is a valid directory stream that has been opened with the `opendir` function.

Return Value

The function returns a valid file descriptor when successful and `-1` when the function is unsuccessful.

See Also

opendir, closedir, readdir

*Example***Listing 3.35 - Using the dirfd function**

```
var
  directoryHandle: PDirectoryStream;
  dFD: integer;
begin
  directoryHandle := opendir(PChar(FDirectory));
  dFD := dirfd(directoryHandle);
end;
```

fchmod LibC.pas*Syntax*

```
function fchmod(
  FileDes: Integer;
  Mode: LongWord
):Integer;
```

Description

This function is similar to the chmod function, but instead of changing the permissions of a filename, the function changes the permissions of the file represented by a valid open file descriptor.

Parameters

FileDes: This is the file descriptor. This value is returned from a call to the `__open` function.

Mode: This is the collection of permissions that will be set. The value in this parameter should be a collection of values from Table 3.2.

If you are adding or removing permissions to a file, it is best to retrieve the current permissions by using the `fstat` function and then incorporate the changes to the permissions as needed.

Return Value

When successful, this function returns 0. If the function was unsuccessful, the function returns -1 and sets `errno` to an appropriate Linux error code.

See Also

chmod, chown

*Example***Listing 3.36 - Using the fchmod function**

```

var
  fdCurrFile: integer;
begin
  fdCurrFile := open(PChar(OpenDialog1.FileName), 0_RDONLY, S_IRWXU);
  fchmod(fdCurrFile, S_IRWXU or S_IRGRP);
  __close(fcCurrFile);
end;

```

fclean LibC.pas*Syntax*

```

function fclean(
  Stream: PIOFile
): Integer;

```

Description

The fclean function forces all buffered data that has yet to be written to disk to be forced to disk and data about to be read is placed back to the file as if the data was not read. Calling this function is equivalent to calling the fflush function.

Parameters

Stream: This is the handle to an open stream.

Return Value

The function returns 0 when successful and -1 when unsuccessful.

See Also

fflush

*Example***Listing 3.37 - Using the fclean function**

```

var
  dataFile: PIOFile;

begin
  //Clean the file
  fclean(dataFile);
end;

```

fclose LibC.pas***Syntax***

```
function fclose(
  Stream: PIOFile
):Integer;
```

Description

This function closes a stream previously opened with `fopen`.

Parameters

Stream: This is the handle to a valid stream.

Return Value

The function returns 0 when successful and -1 when unsuccessful. The reason the function will most likely fail is if the Stream parameter is invalid.

See Also

`fopen`, `fread`, `fwrite`

Example**Listing 3.38 - Using the `fclose` function**

```
Var
  dBaseFile: PIOFile;
begin
  //Open the file
  dBaseFile := fopen(PChar(strFilename), 'r');

  //Perform operations on the stream
  //...

  //Close the File Stream
  fclose(dBaseFile);
end;
```

fcloseall LibC.pas***Syntax***

```
function fcloseall:Integer;
```

Description

This function will flush any buffered output using `fflush` and then close all the open streams in the current process, including standard input, standard output, and standard error.

Return Value

The result of this function will always be 0.

See Also

fflush, fclose, fopen

Example

Listing 3.39 - Using the fcloseall function

```
//Close all streams
fcloseall;
```

fcntl LibC.pas

Syntax

```
function fcntl(
  Handle: Integer;
  Command: SmallInt;
):Integer; cdecl; varargs;
```

Description

This function performs a special file command against a currently open file descriptor. These commands include the ability to set and retrieve flags for a file descriptor or open file, obtain and release locks on particular regions of a file, and specify which process should receive signal notification of input or output when asynchronous input/output is enabled for a file descriptor.

Parameters

Handle: This is the file descriptor that will have the command performed.

Command: This parameter sets the command to execute on the file descriptor and will be one of the constants listed in Table 3.4.

Table 3.4 - Command constants

Constant	Description
F_DUPFD	This command will make a duplicate of the file descriptor and is equivalent to calling the dup function.
F_GETFD	Using this command will return the value of the file descriptor's flags. These flags currently consist of only FD_CLOEXEC, which will cause the file descriptor to close if any of the exec functions (see Chapter 4).
F_SETFD	This command will set the flags for the file descriptor. Like the F_GETFD option, only the FD_CLOEXEC value is valid.
F_GETFL	Using this command returns the flags that were used when the file descriptor was originally opened using the open function. For a list of the values, see the open function in Table 3.8.
F_SETFL	This command sets the new file opening flags to a value of the first varargs parameter. This value should be how you want the file reopened, such as (ORDWR and O_APPEND). If you are modifying the value of these flags, it is wise to obtain the value first using the F_GETFL command and then modify the resulting flag value with additional options.

Constant	Description
F_SETLK	<p>F_SETLK will obtain a lock on the file descriptor by locking a specific region of the file. The first varargs parameter should be a pointer to a TFLock.</p> <pre> type flock = record l_type: Smallint; l_whence: Smallint; l_start: __off_t; l_len: __off_t; l_pid: __pid_t; end; TFLock = flock; PFLock = ^TFLock; </pre> <p>The l_type field specifies the lock operation to perform and will be one of the following constants: F_RDLCK to create a read lock, F_WRLCK to create a write lock, and F_UNLCK to release a lock.</p> <p>l_whence will be either SEEK_SET, which will set the start of the lock from the increment in the l_start parameter, SEEK_CUR, which will begin the lock from the file descriptor's current read/write position + l_start bytes, or SEEK_END which will begin the lock at l_start bytes from the end of file.</p> <p>l_len specifies the size of the lock from the start of the lock. Specifying zero (0) for this field will result in the lock ending at the end of the file.</p> <p>l_pid sets which process owns the lock; only this process has the ability to write to the lock. When the file descriptor is closed or the process ends, the lock is released. When creating a lock, you can ignore filling this in but when checking for locks you must supply the process ID.</p>
F_GETLK	<p>The F_GETLK command requests locking information for a region of the file descriptor. The information is passed in using a pointer to a TFLock. If there is more than one lock in the region that is specified in the TFLock, then the populated result of the TFLock record will contain information about the first lock only.</p>
F_SETLKW	<p>This is equivalent to using the F_SETLC command but will wait until the lock becomes available.</p>
F_GETOWN	<p>The F_GETOWN command returns the process ID that receives the SIGIO signal within an application when the file descriptor is using asynchronous communication. (For further information on signal handling, see Chapter 5.)</p>
F_SETOWN	<p>This command sets the new process ID that will receive and SIGIO messages when the input is read or written to the file descriptor and the file descriptor is using asynchronous input/output.</p>

varargs: The additional varargs parameters for the fcntl function depend on which command was used. For F_DUPID, the only varargs parameter is the new file descriptor. For F_GETFD, F_SETFD, F_GETFL, and F_SETFL, the only varargs parameter represents the flags being sent and received. For the lock operations, the only varargs parameter is a pointer to a TFLock record, which contains the lock information to set or retrieve. For F_GETOWN and F_SETOWN, the additional parameter is the process ID of the input/output owner.

Return Value

The return value for the function is dependent on which command was executed and should be taken from the following table. Those commands not listed in the table do not return a result.

Table 3.5 - Return values for the Command parameter

Command	Description
F_DUPFD	This command returns the new file descriptor when successful or -1 if unsuccessful.
F_GETFD	This command returns the descriptor flags when successful or -1 when unsuccessful.
F_SETFD	This command returns -1 when the function fails. Any other value indicates that the function was successful.
F_GETFL	This command will return the flags when successful and -1 when the command fails.
F_SETLK, F_SETLCKW	If the lock is already set by someone else, the function returns -1; otherwise the function is successful.
F_GETOWN	This command returns the process ID of the file descriptor owner.
F_GETSIG	This is the signal value that is used to indicate the original value that will be retrieved when the function is returned.

See Also

open, __close



Note: When a process terminates abnormally, such as using the abort function, it may leave the lock still active. Using file record locks only works effectively if every process to the file uses locks, since other processes can write over the file. Locking is more a convention to adopt rather than a golden rule. The available options for your application should be to either provide only a single point of access to the processes that manage the record lock or make sure that every process uses the locking system.

fdopen LibC.pas

Syntax

```
function fdopen(  
  FileDes: Integer;  
  Modes: PChar  
):PIOFile;
```

Description

This function returns a stream from an open file descriptor. This function is normally used when you want to use file stream functions where you may only have a file descriptor.

Parameters

FileDes: This is a valid file descriptor that has been opened by a call to `open`.

Modes: This is the method that is used to open the file. The value here is the same for the `fopen` function.

Using `r` as the parameter means that you will be opening the file for read-only access.

Using `w` will open the file for writing and will create the file if it is not already created.

If the file is created, the length of the file will be set to zero.

Using `r+` or `w+` will make the file be open for both read and write access.

When you use any of `r`, `r+`, `w`, or `w+`, once the file has been opened the file's read/write position will be set to the start of the file.

Using `a` for the Modes parameter will open the file that exists for writing and the file's position will be set at the end of the file. If the file does not exist, it will be created.

Like the `a` mode, using `a+` will open the file for appending but will allow both reading and writing to the file. The file will also be positioned at the end of the file when opened.

Return Value

Upon success, this function returns a valid stream. If the function fails, the return value is `nil`.

See Also

`fopen`, `fileno`

Example

Listing 3.40 - Using the `fdopen` function

```
var
  fileStr: PIOFile;
  filedес: integer;
  TextContent: array[0..50] of char;
  PTextContent: PChar;
begin
  if OpenFileDialog1.Execute then
    begin
      edFilename.Text := OpenFileDialog1.Filename;

      //Do a simple operation on the file
      filedес := open(PChar(OpenFileDialog1.Filename), 0_RDONLY);

      //Create the new file stream from the file descriptor
      fileStr := fdopen(filedес, 'r');

      //Read using the file descriptor
      PTextContent := @TextContent;
      fread(PTextContent, sizeof(TextContent), 1, fileStr);
      Memo1.Lines.Add('Read in data: ' + StrPas(TextContent) + '');

      //Close the Stream
      fclose(fileStr);

      //Close the file descriptor
```

```

        __close(filedes);
    end;
end;

```

feof, feof_unlocked LibC.pas

Syntax

```

function feof(
  Stream: PIOFile
):Integer;

function feof_unlocked(
  Stream: PIOFile
):Integer;

```

Description

This function tests if a read/write position of the stream is at the end of the stream.

Parameters

Stream: This is a valid handle to the stream.

Return Value

The feof function returns a non-zero value if it is at the end of the stream and 0 if it is not.

The feof_unlocked function is similar to the feof function with the exception that it should only be called when using file locking with techniques such as fopen_unlocked.

See Also

clearerr, fopen, fclose

Example

Listing 3.4I - Using the feof function

```

procedure TForm1.Button3Click(Sender: TObject);
var
  SourceStream: PIOFile;
  DestStream: PIOFile;
  c: integer;
begin
  //Copy the file by copying individual bytes.

  //Open the Source file
  SourceStream := fopen(PChar(edSourceFile.Text), 'r');

  //Open the destination file (create if needed)
  DestStream := fopen(PChar(edDestFile.Text), 'w+');

  //Copy the individual bytes
  while feof(SourceStream) = 0 do
    begin
      c := getc(SourceStream);
    end;
  end;
end;

```

```

    putc(c, DestStream);
end;

//Close the streams
fclose(DestStream);
fclose(SourceStream);
end;

```

fclose, fclose_unlocked LibC.pas

Syntax

```

function fclose(
  Stream: PIOFile
):Integer;

function fclose_unlocked(
  Stream: PIOFile
):Integer;

```

Description

When reading information from a stream using a function like `fgetc`, if an end of file or error occurs the integer representation of EOF (−1) is returned. You should not assume that the file has reached the end of file. Internally, a stream will store an eof bit and an error bit. By calling the `fclose` function you can determine if the error bit is actually set.

Parameters

Stream: This is the stream to be examined to see if it has the error flag set.

Return Value

This function returns 0 if the value is not set and a non-zero value if the function is set.

See Also

`feof`, `fread`, `fgetc`

Example

Listing 3.42 - Using the `fclose` function

```

if fclose(SourceStream) <> 0 then
begin
  showmessage('A problem occurred while reading the file.');
```

```
  clearerr(SourceStream);
```

```
end;
```

fflush, fflush_unlocked LibC.pas

Syntax

```

function fflush(
  Stream: PIOFile
):Integer;

```

```
function fflush_unlocked(
  Stream: PIOFile
):Integer;
```

Description

The fflush function writes all the buffered output to a valid stream or all output streams. When writing to files, there is some delay while the operating system writes information out to a file. Calling this function forces the operating system to write all the buffered output to the file. You would call this function when you want to be sure you have all the required information on a file before proceeding to another call on the file.

The fflush_unlocked function is similar to the fflush routine, but should only be used when the stream was opened with the fopen_unlocked function.

Parameters

Stream: This parameter will either be a reference to a valid stream or it will be nil to flush all open output streams.

Return Value

This function returns 0 when successful; otherwise, an error occurred. The most likely error returned from errno is EBADF, which is a result of the Stream parameter not being a valid stream.

See Also

fcloseall

Example

Listing 3.43 - Using the fflush function

```
Var
  CurrentStream: PIOFile;
begin
  //Flush a single stream
  fflush(CurrentStream);
  //Flush all streams
  fflush(nil);
end;
```

fgetc, fgetc_unlocked *LibC.pas*

Syntax

```
function fgetc(
  Stream: PIOFile
):Integer
```

```
function fgetc_unlocked(
  Stream: PIOFile
):Integer
```

Description

This function reads a single character from an input stream.

Parameters

Stream: This parameter is a valid input stream.

Return Value

The function returns the character read from the input stream when successful or `-1` when the input stream is at the end of file or has an error. You should examine the `errno` function when the function returns `-1`.

See Also

`getc`, `getchar`, `ungetc`

*Example***Listing 3.44 - Using the `fgetc` function**

```
procedure TForm1.Button3Click(Sender: TObject);
var
  SourceStream: PIOFile;
  DestStream: PIOFile;
  c: integer;
begin
  //Copy the file by copying individual bytes.

  //Open the Source file
  SourceStream := fopen(PChar(edSourceFile.Text), 'r');

  //Open the destination file (create if needed)
  DestStream := fopen(PChar(edDestFile.Text), 'w+');

  //Copy the individual bytes
  while feof(SourceStream) = 0 do
  begin
    c := fgetc(SourceStream);
    putc(c, DestStream);
  end;

  //Close the streams
  fclose(DestStream);
  fclose(SourceStream);
end;
```

fgetpos*, *fgetpos64* LibC.pasSyntax*

```
function fgetpos(
  Stream: PIOFile;
  var Pos: Integer
):Integer;
```

```
function fgetpos64(
  Stream: PIOFile;
  var Pos: Int64
):Integer;
```

Description

The `fgetpos` function returns the stream's current read/write position from the beginning of the stream.

Parameters

Stream: This is a valid file stream.

Pos: This parameter is where the file position will be returned to. This value will be the stream's current position, so that later it can be restored by a call to the `fsetpos` function.

Return Value

When successful, this function returns 0. If not, the function returns `-1` and a valid error code will be returned by the `errno` function.

See Also

`fsetpos`, `ftell`, `fseek`

Example

Listing 3.45 - Using the `fgetpos` function

```
var
  dataFile: PIOFile;
  dataFile_bookmark: fpos_t;

procedure TfrmMain.btnSaveBookmarkClick(Sender: TObject);
begin
  //Get the current Position of the file
  fgetpos(dataFile, dataFile_bookmark);
end;
```

fgets, fgets_unlocked **LibC.pas**

Syntax

```
function fgets(
  S: PChar;
  N: Integer;
  Stream: PIOFile
):PChar;

function fgets_unlocked(
  S: PChar;
  N: Integer;
```



```
Stream: PIOFile
):PChar;
```

Description

The `fgets` function reads a line of text from a file stream. The function will read characters from the stream until it has reached either a new line or the end of the file.

Parameters

S: This is a character buffer that holds the location of where the string will be written to. When a line has been read in from the file stream, if the stream was read to an end of line, the line minus the end of line marker is returned in this buffer.

N: This is the size of the buffer passed in the S parameter and is used to determine the maximum number of characters that can be returned to the string.

Stream: This is the file stream from which the data file will be read.

Return Value

The return value of this function is the same value of the buffer if the function was successful. If the function was at the end of file or the function was unsuccessful, then the return value is nil. If you do receive nil as a result of the function, you should make a call to `errno` to see if the function returned an error.

See Also

`getline`, `getdelim`

Example

Listing 3.46 - Using the `fgets` function

```
procedure TForm1.Button2Click(Sender: TObject);
var
  ln: array[0..500] of char;
begin
  //Create the file if needed
  if dataFile = nil then
    begin
      dataFile := fopen(PChar(edFilename.text), 'r');
      linepos := 1;
    end;

  //Read the Line in
  fgets(ln, 500, dataFile);

  //Display the Line
  mmText.Lines.Add(inttostr(linepos)+' '+ln);
  inc(linepos);
end;
```

fileno, fileno_unlocked LibC.pas***Syntax***

```
function fileno(
  Stream: PIOFile
):Integer

function fileno_unlocked(
  Stream: PIOFile
):Integer
```

Description

This function will return a valid file descriptor ID when a stream is passed in as a parameter. It is often useful when you have an open stream and you have particular functions that only work with file descriptors.

Parameters

Stream: The Stream parameter is a valid stream that was created by the fopen function or any other function that creates a valid stream.

Return Value

If successful, the function returns a valid file descriptor. If the function was unsuccessful, it returns -1. One reason the function could fail is that the value passed to the Stream parameter was not a valid stream.

See Also

fdopen

Example**Listing 3.47 - Using the fileno function**

```
procedure TForm1.btnBrowseClick(Sender: TObject);
var
  fileStr: PIOFile;
  filedes: integer;
  TextContent: packed array[0..50] of char;
begin
  if OpenFileDialog1.Execute then
  begin
    edFilename.Text := OpenFileDialog1.Filename;
    //Do a simple operation on the file
    FileStr := fopen(PChar(OpenDialog1.Filename), 'r');

    //Create the new file descriptor
    filedes := fileno(fileStr);

    //Read using the file descriptor
    __read(filedes, TextContent, sizeof(TextContent));
```

```

    Memo1.Lines.Add('Read in data: ' + StrPas(TextContent) + '');

    //Close the file descriptor
    __close(filedes);

    //Close the Stream
    fclose(FileStr);
end;
end;

```

fopen, fopen64 LibC.pas

Syntax

```

function fopen(
  FileName: PChar;
  Modes: Pchar
):PIOFile;

```

```

function fopen64(
  FileName: PChar;
  Modes: Pchar
):PIOFile;

```

Description

The `fopen` function will open the file `FileName` and return a valid stream. The `fopen64` function is equivalent to the `fopen` function but deals with files larger than $2^{32}-1$ bits.

Parameters

FileName: This is the name of the file to open.

Modes: The `Modes` parameter specifies how the file will be opened. The following modes are valid for the `fopen` function.

Using `r` as the parameter means that you will be opening the file for read-only access.

Using `w` will open the file for writing and will create the file if it is not already created.

If the file is created, the length of the file will be set to zero.

Using `r+` or `w+` opens for both read and write access.

When you use `r`, `r+`, `w`, or `w+`, once the file has been opened, the file's read/write position will be sent to the start of the file.

Using `a` for the `Modes` parameter will open a file that already exists for writing and the file's position will be set at the end of the file. If the file does not exist, it will be created.

Like the `a` mode, using `a+` will open the file for appending but will allow both reading and writing to the file. The file, when opened, will also be positioned at the end of the file.

Return Value

If `fopen` succeeds, the function returns a valid file stream. If the function fails, the function will return `nil` and calling `errno` will return the error description. The error will often be due to a lack of permissions (`errno` returns `EACCES`) or because an invalid file was opened (`errno` returns `EINVAL`).

See Also

open, fclose, fdopen, freopen

Example

Listing 3.48 - Using the fopen function

```
//Open the file for reading
CurrentFile := fopen(PChar(edFilename.Text), 'r');
```

fprintf LibC.pas

Syntax

```
function fprintf(
  Stream: PIOFile;
  Format: PChar
):Integer; cdecl; varargs;
```

Description

This function writes a formatted string to a stream. The formatted string is based on a combination of the Format parameter with additional varargs parameters. Using this function allows an easy way to separate the content of a string with the format of the resulting string.

Parameters

Stream: This is the stream that will have the formatted string sent to it.

Format: This is the format of how the information will be presented and uses a simple convention for writing out simple values. Whenever a percentage symbol (%) is found in the Format string, the next character determines the type of data that will be inserted into the character’s position.

Common data types of the symbol delimiters include those listed in Table 3.6.

Table 3.6 - Common tokens for the fprintf function

Delimiter	Data Type	Example
%d %i	Integer	//Write a number fprintf(dataFile, 'The number is %d'#13#10, 1097);
%x %X	Unsigned integer	//Write out a hexadecimal value SomeIntegerValue := 237; fprintf(dataFile, 'The hex number for %d is %x'#13#10, SomeIntegerValue, SomeIntegerValue);
%e %E	Double	//Write an extended number SomeDoubleValue := pi; fprintf(dataFile, 'An extended value is %E'#13#10, SomeDoubleValue);

Delimiter	Data Type	Example
%f %F	Double	//Write out a floating point value SomeDoubleValue := pi * 9 * 9; fprintf(dataFile, 'The circumference is %f cm'#13#10, SomeDoubleValue);
%s	PChar	//Write out a string to file StrPCopy(SomePChar, 'Hello from a string'); fprintf(dataFile, 'The message is: %s', SomePChar);
%%		//Write out a percentage symbol SomeDoubleValue := 2950 * 0.10; fprintf(dataFile, '10%% of 2950 is %f'#13#10, SomeDoubleValue);

varargs: The varargs list contains the corresponding values that will be read from the format string. For example, if there are three symbols used in the format string, there should be three entries in the varargs list and the data types of the varargs list should match the data types passed into the Format string.

Return Value

When successful, this function returns the number of characters that could have been successfully written to the stream. If the function fails, the return value is -1.

See Also

fputs

fputc, fputc_unlocked, putc *LibC.pas*

Syntax

```
function fputc(
  C: Integer;
  Stream: PIOFile
):Integer
```

```
function fputc_unlocked(
  C: Integer;
  Stream: PIOFile
):Integer;
```

```
function putc(
  C: Integer;
  Stream: PIOFile
):Integer;
```

Description

The fputc function writes an individual character to an open stream.

Parameters

C: This is an integer value that represents the ordinal value of the character to be written to the stream. You should use the Pascal `ord` function to get the individual character.

Stream: This is the input/output stream that is written to.

Return Value

This function returns the ordinal value of the character that was written to the stream when successful. If the function was unsuccessful, the file will return the integer value `EOF`.

See Also

`fgetc`, `fputs`

Example

Listing 3.49 - Using the `fputc` function

```
procedure TForm1.Button3Click(Sender: TObject);
var
  SourceStream: PIOFile;
  DestStream: PIOFile;
  c: integer;
begin
  //Copy the file by copying individual bytes.

  //Open the Source file
  SourceStream := fopen(PChar(edSourceFile.Text), 'r');

  //Open the destination file (create if needed)
  DestStream := fopen(PChar(edDestFile.Text), 'w+');

  //Copy the individual bytes
  while feof(SourceStream) = 0 do
  begin
    c := fgetc(SourceStream);
    fputc(c, DestStream);
  end;

  //Close the streams
  fclose(DestStream);
  fclose(SourceStream);
end;
```

fputs, fputs_unlocked **LibC.pas**

Syntax

```
function fputs(
  const S: PChar;
  Stream: PIOFile
):Integer;
```

```
function fputs_unlocked(
  const S: PChar;
  Stream: PIOFile
):Integer;
```

Description

This function writes a string S into a file stream. The function writes the string without placing the end of string character in the stream.

Parameters

S: This is the string that will be written to the stream.

Stream: This is where the string will be written to.

Return Value

When successful, the function returns a valid positive integer. If the function fails, it returns -1.

See Also

fgets, fopen, fputc

Example**Listing 3.50 - Using the fputs function**

```
procedure TForm1.Button1Click(Sender: TObject);
var
  memStr: PIOFile;
  Buffer: PChar;
  BufferSize: integer;
begin
  //Create the Memory Stream
  memStr := open_memstream(@Buffer, @BufferSize);

  //Write into the Buffer
  fprintf(memStr, 'The next random number is %d'#10, random(100));
  fputs('This is an extra line placed in the buffer'#10, memStr);

  //Close the stream. This will also push the data from
  //the stream into the Buffer.
  fclose(memStr);

  mmLines.Lines.Add(StrPas(Buffer));
end;
```

fread, fread_unlocked LibC.pas**Syntax**

```
function fread(
  Ptr: Pointer;
  Size: Integer;
```

```

N: LongWord;
Stream: PIOFile
):LongWord;

function fread_unlocked(
Ptr: Pointer;
Size: Integer;
N: LongWord;
Stream: PIOFile
):LongWord;

```

Description

The `fread` function will read a block or set of blocks of data in from a stream.

Parameters

Ptr: This value is a pointer to where the information that is read from the stream will be stored. Normally you would pass the address of a variable into this parameter by using the `@` operator on a variable (i.e., `@MyInteger`).

Size: This is the size of the block of data that you wish to read in. In most cases this will be the size of a record or simple data type. It is recommended that you use the `sizeof` function against the data type such as `sizeof(TCustomerRecord)` or `sizeof(integer)`.

N: This is the number of blocks that you wish to read in from the stream. This parameter is very handy when the `Ptr` parameter points to an array and you wish to read several elements into the array.

Stream: This is the input/output stream from which the data is read.

Return Value

If successful, the function returns how many blocks of data were read into the memory at the `Ptr` parameter. If the function fails, it returns 0.

See Also

`fopen`, `fwrite`, `fwrite_unlocked`, `fclose`

Example

Listing 3.51 - Using the `fread` function

```

//Demo to read in the header information from a dbase file.
type
  TdBaseHeader = packed record
    InfoByte: byte;
    LastUpdateYear: byte;
    LastUpdateMonth: byte;
    LastUpdateDay: byte;
    NumberOfRecords: dword;
    BytesInTheHeader: word;
    BytesInTheRecord: word;
    Reserved1: word;

```



```

    InTransaction: byte;
    EncryptionFlag: byte;
    Reserved2: array[0..11] of char;
    HasMDXFile: byte;
    LanguageDriver: byte;
    Reserved3: word;
end;

var
    dBaseFile: PIOFile;
    FFileHeader: TdBaseHeader;

procedure OpenDBaseFile;
begin
    //Read in the header
    dBaseFile := fopen(PChar(Filename), 'r');
    fread(@FFileHeader, sizeof(TdBaseHeader), 1, dBaseFile);
end;

```

***freopen, freopen64* LibC.pas**

Syntax

```

function freopen(
    FileName: PChar;
    Modes: PChar;
    Stream: PIOFile
):PIOFile;

function freopen64(
    FileName: PChar;
    Modes: PChar;
    Stream: PIOFile
):PIOFile;

```

Description

The `freopen` function closes an existing stream and then returns a new file stream based on the `FileName` and `Modes` parameters. The function call is equivalent to calling the following:

```

fclose(Stream);
Result := fopen(Filename, Modes, 0);

```

The `freopen` function is especially useful when redirecting standard input, standard output, and standard error in console applications.

Parameters

FileName: This is the filename of the file that will be the replacement of the stream.

Modes: This is how the file will be opened and is the same value used for the `Modes` parameter in the `fopen` function.

Stream: This is an existing stream that will be closed.

Return Value

The function returns a valid stream when successful and nil if the function was unsuccessful.

See Also

fopen, fclose

Example

Listing 3.52 - Using the freopen function

```
procedure TfrmMain.btnRedirectOutputClick(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    //Redirect Standard Output to a file
    dataFile := freopen(PChar(SaveDialog1.FileName), 'w+',
      fdopen(STDOUT_FILENO, 'w'));
  end;
end;
```

fseek, fseeko, fseeko64 *LibC.pas*

Syntax

```
function fseek(
  Stream: PIOFile;
  Off: Integer;
  Whence: Integer
):Integer;
```

```
function fseeko(
  Stream: PIOFile;
  Off: Integer;
  Whence: Integer
):Integer;
```

```
function fseeko64(
  Stream: PIOFile;
  Off: Int64;
  Whence: Integer
):Integer;
```

Description

The fseek function will reposition the stream's input/output position so that the next data read or written will be from that position.

The fseeko function is functionally equivalent to the fseek function, but the value from the Whence parameter is an integer typecast of the off_t data type, which is used for

compatibility against older GLibC libraries. On the Linux systems tested, the `fseeko` function always returned the same value as the `fseek` function.

Parameters

Stream: This is the stream that is to be repositioned. Not using a valid stream in this parameter will cause the function to fail and the `errno` value will be set to `EBADF`.

Off: This is the position of where to move to. This parameter works in conjunction with the **Whence** parameter to determine the position of the stream.

Whence: This parameter will be either the constant value `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`. Using `SEEK_SET` will reposition the stream's internal position to `Off` bytes from the beginning of the file. `SEEK_CUR` will reposition the stream's position `Off` bytes from the stream's current read/write position. The `SEEK_END` constant will make the stream's read/write position `Off` bytes from the end of the file. Using any value except `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` will result in the function failing and the `errno` value being set to `EINVAL`.

Return Value

When successful, the function returns 0. If the function was unsuccessful, the return value is `-1` and the appropriate error is stored in `errno`.

See Also

`ftell`, `lseek`

Example

Listing 3.53 - Using the `fseek` function

```
//Go to the start of where the data is
//which is located after the header record
fseek(dBaseFile, FFileHeader.BytesInTheHeader, SEEK_SET);
```

`fsetpos`, `fsetpos64` LibC.pas

Syntax

```
function fsetpos(
  Stream: PIOFile;
  const Pos: PFPos
):Integer;

function fsetpos64(
  Stream: PIOFile;
  const Pos: PFPos64
):Integer;
```

Description

The `fsetpos` function sets a stream's read/write position to a fixed number of bytes from the beginning of the stream. This function is equivalent to using the `fseek` function with the

SEEK_SET used as the mode. The fsetpos64 is used when dealing with files larger than $2^{32}-1$ bytes.

Parameters

Stream: This is the stream whose read/write position will be moved. If this parameter does not point to a valid stream, the function will fail and errno will be set to EBADF.

Pos: This is a pointer to a variable that tells the function how many bytes from the start to move by. After the function has been called, the value referenced at this position tells the actual position of the stream, which may be different from the position that you intended to move to.

Return Value

Upon success, the function returns 0. If the function fails, -1 will be returned.

See Also

fseek, ftell, ftello, ftell64

Example

Listing 3.54 - Using the fsetpos function

```
var
    dataFile: PIOFile;
    dataFile_bookmark: fpos_t;

procedure TfrmMain.btnGotoBookmarkClick(Sender: TObject);
begin
    //Goto the Bookmark position
    fsetpos(dataFile, @dataFile_bookmark);
end;
```

fstat, fstat64 *LibC.pas*

Syntax

```
function fstat(
    FileDes: Integer;
    var StatBuffer: TStatBuf
):Integer;

function fstat64(
    FileDes: Integer;
    var StatBuffer: TStatBuf64
):Integer;
```

Description

The fstat function returns information about a given file. This is similar to the stat function but instead of retrieving file information from a filename, the information is obtained by passing an open file descriptor.

Parameters

FileDes: This is a valid file descriptor that has been opened with the `open` function or another function that returns a valid file descriptor.

StatBuffer: This is a `TStatBuf` variable that contains details about the file. For more information about the file information, examine the section titled “Getting Information and Attributes about a File” earlier in this chapter.

Return Value

This function returns 0 when successful and `-1` when unsuccessful. The main reason that `fstat` would fail is due to the `FileDes` parameter being an invalid file descriptor.

See Also

`open`, `stat`, `lstat`

Example

Listing 3.55 - Using the `fstat` function

```
procedure TForm1.Button1Click(Sender: TObject);
var
    fd: integer;
    buf: TStatBuf;
begin
    if OpenFileDialog1.Execute then
    begin
        fd := open(PChar(OpenDialog1.FileName), O_RDONLY, S_IRWXU);
        fstat(fd, buf);

        //Examine the contents of buf

        __close(fd);
    end;
end;
```

ftell, ftello, ftello64 **LibC.pas**

Syntax

```
function ftell(
    Stream: PIOFile
):Integer;
```

```
function ftello(
    Stream: PIOFile
):Integer;
```

```
function ftello64(
    Stream: PIOFile|
):Int64;
```

Description

The `ftell` function will return the number of bytes from the beginning of a stream.

In the `ftello` function, although semantically equivalent to `ftell`, the resulting value is actually a data type known as `t_off` which may not be an integer value but a pointer to another structure.

Parameters

Stream: This is a valid stream.

Return Value

If successful, this function returns the position from the start of the stream. The returned value can then be used to set the position back with the `fseek` function when the value was retrieved by `ftell`.

If the function was unsuccessful, it returns `-1`. The most likely reason that `ftell` would fail is if the value passed into the Stream parameter was not a valid stream.

See Also

`ftello`, `ftello64`, `fseek`

Example

Listing 3.56 - Using the `ftell` function

```

type
  TForm1 = class(TForm)
  ...
  public
    CurrentFile: PIOFile;
  end;

procedure TForm1.btnTellClick(Sender: TObject);
var
  currPos: integer;
begin
  currPos := ftell(CurrentFile);
  lblPosition.Caption := inttostr(currPos);
end;

```

`fwrite`, `fwrite_unlocked` LibC.pas

Syntax

```

function fwrite(
  const __ptr: Pointer;
  Size: Integer;
  N: LongWord;
  Stream: PIOFile
):LongWord;

```

```
function fwrite_unlocked(
const Ptr: Pointer;
Size: Integer;
N: LongWord;
Stream: PIOFile
):LongWord;
```

Description

The fwrite function writes a number of blocks of a defined size out to an open stream.

Parameters

`__ptr`: This is a pointer to where the data that is to be written to the file stream is located.

`Size`: This is the size of the block that will be written to the file stream. This parameter should work in conjunction with the `N` parameter to write out the correct number of blocks.

`N`: This is the number of blocks that will be written to the file stream. The total amount of data that will be written to the file stream will be `Size * N` bytes.

`Stream`: This is a valid open input enabled stream.

Return Value

This function returns the number of blocks defined in the `N` parameter that were successfully written to the file stream. If the function was unsuccessful, it returns `-1` and the appropriate error code will be returned on the next call to `errno`.

See Also

`fread`, `fopen`, `fclose`

Example

Listing 3.57 - Using the fwrite function

```
type
//Structure that holds the information for the dBase file
TdBaseHeader = packed record
    InfoByte: byte;
    LastUpdateYear: byte;
    LastUpdateMonth: byte;
    LastUpdateDay: byte;
    NumberOfRecords: dword;
    BytesInTheHeader: word;
    BytesInTheRecord: word;
    Reserved1: word;
    InTransaction: byte;
    EncryptionFlag: byte;
    Reserved2: array[0..11] of char;
    HasMDXFile: byte;
    LanguageDriver: byte;
    Reserved3: word;
end;
```

```

var
    dBaseFile: PIOFile;
    FFileHeader: TdBaseHeader;

begin
    dBaseFile := fopen(PChar(strFilename), 'w+');
    //Write the contents of the dBase file out to disk.
    //Write the Header
    fwrite(@FFileHeader, sizeof(TdBaseHeader), 1, dBaseFile);

    //Do more writing operations on the file
    fclose(dBaseFile);
end;

```

get_current_dir_name* *LibC.pas

Syntax

```
function get_current_dir_name: PChar;
```

Description

This function returns the application's current working directory.

Return Value

This function returns a pointer to an existing allocated string buffer that contains the application's working directory.

See Also

`__chdir`, `getcwd`, `getwd`

Example

Listing 3.58 - Using the `get_current_dir_name` function

```

showmessage('The Current Working Directory is: '+get_current_dir_name);
__chdir('/tmp');
showmessage('The Current Working Directory is: '+get_current_dir_name);

```

getc, getc_unlocked* *LibC.pas

Syntax

```

function getc(
    Stream: PIOFile
):Integer;

function getc_unlocked(
    Stream: PIOFile
):Integer;

```


Description

The `getc` function returns a single character from a valid stream and returns the character typecast as an integer.

Parameters

Stream: This is the input stream from which the character will be read.

Return Value

The function returns the read-in character typecast to an integer when successful. If the function returns `-1`, either the function has reached the end of file or an error has occurred while reading the character. You should make a call to `errno` to determine whether the function did, in fact, reach the end of file or had an error.

See Also

`putc`, `fgets`, `fputs`

*Example***Listing 3.59 - Using the `getc` function**

```
procedure TForm1.Button3Click(Sender: TObject);
var
  SourceStream: PIOFile;
  DestStream: PIOFile;
  c: integer;
begin
  //Copy the file by copying individual bytes.

  //Open the Source file
  SourceStream := fopen(PChar(edSourceFile.Text), 'r');

  //Open the destination file (create if needed)
  DestStream := fopen(PChar(edDestFile.Text), 'w+');

  //Copy the individual bytes
  while feof(SourceStream) = 0 do
  begin
    c := getc(SourceStream);
    putc(c, DestStream);
  end;

  //Close the streams
  fclose(DestStream);
  fclose(SourceStream);
end;
```

getcwd LibC.pas*Syntax*

```
function getcwd(
  Buffer: PChar;
```

BufSize: Integer
):PChar;

Description

This function fills a string buffer with the information about the current working directory of the application.

Parameters

Buffer: This is either a pointer to an existing allocated string buffer or nil. If the value is nil, the memory will be allocated by the function call and you will be responsible for releasing any allocated memory.

BufSize: This is the size of the Buffer in bytes that will be written to when the Buffer parameter is allocated and is zero when nil has been passed as the parameter to Buffer.

Return Value

The function returns nil if the function fails; otherwise, the return value is the same as the Buffer parameter.

See Also

getwd, get_current_dir_name

Example

Listing 3.60 - Using the getcwd function

```
procedure TForm1.FormCreate(Sender: TObject);
var
  dirName: array[0..300] of char;
begin
  //Get the Directory
  getcwd(dirName, 300);
  edDirectory.Text := dirName;
end;
```

getdelim *LibC.pas*

Syntax

```
function getdelim(
  LinePtr: PPChar;
  N: PLongWord;
  Delimiter: Integer;
  Stream: PIOFile
):Integer;
```

Description

This function reads characters from a file until it reaches a specific delimiter character. It is similar to using the fgets function, but instead of reading until the end of line, it reads until the delimiter character is found or the end of file is reached.

Parameters

LinePtr: This is a pointer to a buffer containing a string buffer. This parameter works in conjunction with the **N** parameter to specify how the buffer is managed. If the **LinePtr** is a pointer to an actual buffer of **N** characters and up to **N** characters are read in, this parameter will hold the string. On the other hand, if there is insufficient space in the buffer for the text, then new memory will be allocated and the pointer will point to the newly created buffer.

Setting the **LinePtr** parameter to **nil** and the **N** parameter to 0 also results in the memory for the read-in content to be allocated and returned in the buffer.

N: When calling the function, the value pointed to at **N** holds the number of characters in the buffer. Upon returning from the function, this parameter will hold the number of characters in the buffer as the buffer size may have changed due to insufficient space.

Delimiter: This is the character that will be used as a delimiter. It is best to use the **ord** function to obtain an integer representation of the character; for example, **ord('<')**.

Stream: This is the stream from which the data is being read.

Return Value

When successful, the function returns the number of characters read from the stream. When unsuccessful or at the end of the stream, the function returns **-1**.

See Also

getline, **fgets**

Example

Listing 3.61 - Using the **getdelim** function

```
procedure TForm1.Button2Click(Sender: TObject);
var
  f: PIOFile;
  sText: array[0..500] of char;
  info: PChar;
  RetStrLen: LongWord;
begin
  //A simple application to extract the XML Tags from an XML Doc
  if not FileExists(edFilename.Text) then
  begin
    MessageDlg('Please select a file before proceeding.',
      mtConfirmation, [mbOK], 0);
    Exit;
  end;

  info := sText;
  //Open the file
  f := fopen(PChar(edFilename.Text), 'r');

  //Read in the XML Tags
  while feof(f) = 0 do
  begin
```

```

//Read upto the '<' Tag
getdelim(@info, @RetStrLen, ord('<'), f);
//Read upto the '>' Tag
getdelim(@info, @RetStrLen, ord('>'), f);

//Display what's in the Tag
mmLines.Lines.Add('<'+StrPas(info));
end;

//Close the file
fclose(f);
end;

```

getline LibC.pas

Syntax

```

function getline(
  LinePtr: PChar;
  N: PLongWord;
  Stream: PIOFile
):Integer;

```

Description

The `getline` function reads a line from a valid stream and returns a string that represents the line that was read from the stream, a new line, and a null-terminating character (#0).

Parameters

LinePtr: This is a pointer to a PChar string buffer. This parameter works in conjunction with the **N** parameter to specify how the buffer is managed. If the **LinePtr** is a pointer to an actual buffer of **N** characters and up to **N** characters are read in, this parameter will hold the string. On the other hand, if there is insufficient space in the buffer for the text, new memory will be allocated and the pointer will point to the newly created buffer.

Setting the **LinePtr** parameter to `nil` and the **N** parameter to 0 also results in the memory for the read-in content to be allocated and returned in the buffer.

N: When calling the function, the value pointed to at **N** holds the number of characters in the buffer. Upon returning from the function, this parameter will hold the number of characters in the buffer as the buffer size may have changed due to insufficient space.

Stream: This is the stream from which the data is being read.

Return Value

When successful, the function returns the number of characters read from the stream. When unsuccessful or at the end of the stream, the function will return `-1`.

See Also

`getdelim`, `fgets`

*Example***Listing 3.62 - Using the getline function**

```

procedure TfrmMain.btnGetLineClick(Sender: TObject);
var
  f: PIOFile;
  sText: array[0..500] of char;
  Lineinfo: PChar;
  RetStrLen: LongWord;
  LineCounter: integer;
begin
  if not FileExists(edFilename.Text) then
    begin
      MessageDlg('Please select a file before proceeding.',
        mtConfirmation, [mbOK], 0);
      Exit;
    end;

  LineCounter := 0;

  Lineinfo := sText;
  //Open the file
  f := fopen(PChar(edFilename.Text), 'r');

  //Read in the Lines from the file
  while feof(f) = 0 do
    begin
      inc(LineCounter);

      //Read the line in from the stream.
      getline(@Lineinfo, @RetStrLen, f);

      //Display what's in the Tag
      mmLines.Lines.Add('Line '+inttostr(LineCounter)+': '
        +StrPas(Lineinfo));
    end;

    mmLines.Lines.Add('Read in a total of '+inttostr(LineCounter)+
      ' Lines');

  //Close the file
  fclose(f);
end;

```

getw LibC.pas*Syntax*

```

function getw(
  Stream: PIOFile
):Integer;

```

Description

This function reads a single integer from the currently open stream. This function can potentially lead to confusion when the function returns `-1`, as it either reads in a `-1` value or may be at the end of the stream. The function is in the Linux API for backward compatibility and should not be used. Using the `fread` function is a safer option.

Parameters

Stream: This is the stream that will be read.

Return Value

The function returns the value of the integer that was just read in from the stream. The function will return `-1` if the end of the stream is reached.

See Also

`fread`, `putw`

Example

Listing 3.63 - Using the `getw` function

```
procedure TfrmMain.btnLoadValuesClick(Sender: TObject);
var
  fs: PIOFile;
  value: integer;
begin
  if edFilename.Text <> '' then
    begin
      fs := fopen(PChar(edFilename.Text), 'r');
      //Read in the values
      value := getw(fs);
      mmText.Lines.Add('Value 1 is '+inttostr(value));
      value := getw(fs);
      mmText.Lines.Add('Value 2 is '+inttostr(value));
      value := getw(fs);
      mmText.Lines.Add('Value 3 is '+inttostr(value));
      fclose(fs);
    end;
end;
```

getwd LibC.pas

Syntax

```
function getwd(
  Buf: PChar
):PChar;
```

Description

This function obtains the current working directory of the application and places the value into the `Buf` parameter. This function is similar to the `getcwd` function; however, it does

assume that the buffer has sufficient memory to hold the name of the directory. Using the `getcwd` function is a safer method of obtaining the working directory.

Parameters

Buf: This is the character buffer that will hold the directory when it is returned. Unlike the `getcwd` command, you cannot place `nil` in this parameter to allow the function to allocate sufficient memory.

Return Value

The function returns the `Buf` pointer when successful. If the function is unsuccessful, the result is `nil` and a call to the `errno` function will determine the reason the function was unsuccessful.

See Also

`getcwd`, `get_current_dir_name`

Example

Listing 3.64 - Using the `getwd` function

```
procedure TForm1.FormCreate(Sender: TObject);
var
  dirName: array[0..300] of char;
begin
  //Get the Directory
  getwd(dirName);
  edDirectory.Text := dirName;
end;
```

ioctl ***LibC.pas***

Syntax

```
function ioctl(
  __fd: Integer;
  __request: LongWord
): Integer; cdecl; varargs;
```

Description

The `ioctl` function performs a special operation on an open device. This operation is normally something unique to the device such as playing an audio track on a CD-ROM device or retrieving the position and button status of a joystick device. The function works by passing in the file descriptor of an open device and then a constant to denote which command to perform on the device. If the command that is passed into the `__request` parameter requires additional information, the additional information can be passed into the function through the use of the `varargs` parameter.

Parameters

__fd: This parameter is a valid open file descriptor.

`__request`: This is the specific command to execute on the device. This value is normally unique to the device passed in the `__fd` parameter. In most cases, the constant will not be found in the `LibC.pas` file that ships with Kylix, and in many cases these constants will need to be converted from values found in their original C header files.

`varargs`: Additional parameters can be passed to the `ioctl` function if the command specified in the `__request` parameter requires additional information. The types of values in most cases depend on the command that is to be executed. In most cases, the operations are usually either operations that perform some commands or read/write settings to the device.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return -1 and the error code can be retrieved by using the `errno` function. The most likely reasons for this function to fail are if the file descriptor is invalid, the device does not accept the command passed into the `__request` parameter, or the additional arguments are invalid.

See Also

`open`, `__close`

Example

See Listing 3.20 - Ejecting a CD-ROM device.

lseek, lseek64 ***LibC.pas***

Syntax

```
function lseek(
  Handle: Integer;
  Offset: Integer;
  Direction: Integer
):Integer;
```

```
function lseek64(
  FileDes: Integer;
  Offset: Int64;
  Whence: Integer
):Int64;
```

Description

This function will position the file to begin reading and writing from a particular position. The position can be located from the start of the file, end of the file, or the file's current position. This function is only practical when used with files and not with interprocess methods such as pipes or named pipes.

Parameters

`Handle`: This is the handle of the file descriptor.

Offset: This is an integer count of how many bytes the file's read/write position will move. This value works in conjunction with the Direction parameter to determine where to move the file.

Direction: This parameter is one of the following integer constants in Table 3.7.

Table 3.7 - Options for the lseek function

Constant	Meaning
SEEK_SET	Using SEEK_SET will move the file's position Offset bytes from the beginning of the file.
SEEK_CUR	Using SEEK_CUR will move the file's position Offset bytes from the file's current read/write position.
SEEK_END	Using SEEK_END will move the file's position Offset bytes from the end of the file.

Return Value

This function returns 0 when successful and -1 when unsuccessful. When the function is unsuccessful, the errno function will return EBADF if the Handle parameter is not a valid file descriptor. The errno will return ESPIPE when this function is used against a pipe or named pipe file descriptor. If the Direction parameter is not one of the values SEEK_SET, SEEK_CUR, or SEEK_END, the function will fail and the EINVAL value will be returned.

See Also

fseek, ftell, ftello

Example

Listing 3.65 - Using the lseek function

```
FileHandle := open(PChar(edFilename.Text), 0_RDWR, 0);
lseek(FileHandle, 32, SEEK_SET);
__read(FileHandle, Customer, sizeof(TCustomer));
__close(FileHandle);
```

Istat, Istat64 LibC.pas

Syntax

```
function Istat(
  FileName: PChar;
  var StatBuffer: TStatBuf
):Integer;

function Istat64(
  FileName: PChar;
  var StatBuffer: TStatBuf64
):Integer;
```

Description

This function returns information on a given file. This is similar to the `stat` function, but this function will not traverse a linked file. Instead, if the linked file has been entered as the `FileName` parameter, details about the linked file are returned in the `StatBuffer` parameter. If the file in the `FileName` parameter is not a linked file, then just the information on the file is returned.

Parameters

`FileName`: This is a valid file descriptor.

`StatBuffer`: This is a `TStatBuf` variable that contains details about the file. For more information about the file information, examine the section titled “Getting Information and Attributes about a File” earlier in this chapter.

Return Value

This function returns 0 when successful and -1 when is unsuccessful.

See Also

`stat`, `fstat`

Example

Listing 3.66 - Using the `lstat` function

```
procedure TTKFileListBox.LoadDirectoryEntries;
var
  directoryHandle: PDirectoryStream;
  dFD: integer;
  PDirInfo: PDirEnt;
  sbuf: TStatBuf;
begin
  Items.Clear;
  //Here is where we read the entries of the directory
  directoryHandle := opendir(PChar(FDirectory));
  dFD := dirfd(directoryHandle);

  if directoryHandle <> nil then
    try
      PDirInfo := readdir(directoryHandle);
      while PDirInfo <> nil do
        begin
          //In reality the PDirInfo^.d_type should hold the information
          //if the file is a directory, but this is not implemented
          //in all versions. The stat function is more reliable.

          //Get information about a file and do not follow
          //symbolic links.
          lstat(PChar(FDirectory + '/' + PDirInfo^.d_name), sbuf);
          if sbuf.st_mode and __S_IFDIR > 0 then
            begin
              //This is a directory
              Items.Add([''+StrPas(PDirInfo^.d_name)+'']);
            end else begin
```

```

        //This is a file
        Items.Add(PDirInfo^.d_name);
    end;

    PDirInfo := readdir(directoryHandle);
end;
finally
    closedir(directoryHandle);
end else begin
    raise ETKFileListBoxError.Create('The directory ''' + FDirectory + ''' could '+
    'not be opened.');
```

open, open64 LibC.pas

Syntax

```

function open(
  PathName: PChar;
  Flags: Integer;
  Mode: Integer
):Integer;
```

```

function open64(
  PathName: PChar;
  Flags: Integer;
  Mode: Integer
):Integer;
```

Description

This function returns a new file descriptor for a particular filename, in one of many modes for reading, writing, or appending. This function also enables you to specify permissions for the file when the file is being created.

Parameters

PathName: This is a relative or absolute path to the filename.

Flags: The Flags parameter specifies how the file should be opened. When setting the Flags parameter you have the option of including the bitwise OR of any of the values from Table 3.8.

Table 3.8 - Flags constants used when opening files

Constant	Description
O_RDONLY	This value will open the file for reading.
O_WRONLY	This value will open the file for writing.
O_RDWR	This function will open the file for reading and writing.
O_CREAT	If the file does not exist, including this option will create the file.

Constant	Description
O_EXCL	Including this option when also using O_CREAT results in the function failing if the file that is to be opened already exists.
O_NOCTTY	When using this option, if the file that is being opened is a terminal device, the opened device will not control the process.
O_TRUNC	Including this option will truncate the file when it is opened.
O_APPEND	When the file is opened in write mode, the file is opened and the read/write position is located at the end of the file.
O_NONBLOCK, O_NDELAY	Using these options causes the read or write function to return immediately from the call, returning an error only if they could not perform the read or write operation immediately.
O_FSYNC, O_SYNC, O_DSYNC, O_RSYNC	Using any of these options opens the file for synchronous writing. Under this mode the function will not return until the data is written to the file descriptor.
O_ASYNC	When using this mode, input will be obtained asynchronously and notifications that input is available are handled through signals.
O_DIRECT	Using this option will cause the file to be opened and information to be written directly to disk.
O_DIRECTORY	Using this option causes the function to fail if the filename parameter is not a directory.
O_NOFOLLOW	Using this option will result in the function failing if the file is actually a linked file.

Mode: This parameter specifies the permissions for the file if it is to be created and should be values from Table 3.2. If the function is an existing file, you can simply pass 0 as the parameter for this function.

Return Value

When successful, the function will return a valid positive file descriptor. If the function is unsuccessful, it returns `-1` and an appropriate value will be returned by the `errno` function.

See Also

`fopen`, `__close`, `creat`, `chmod`

Example

Listing 3.67 - Using the open function

```
...
public
{ File Descriptor Handle }
  FileHandle: integer;
end;
...
```

```
begin
    FileHandle := open(PChar(edFilename.Text), 0_RDWR, 0);
end;
```

***open_memstream* LibC.pas**

Syntax

```
function open_memstream(
    BufLoc: PPChar;
    SizeLoc: PLongWord
):PIOFile;
```

Description

This function creates a new buffer to which you can read and write content that is held internally within memory. When the stream is closed or flushed, the contents of the stream will be copied to internal memory of the application.

Parameters

BufLoc: This is the buffer the stream is written to when the stream is either closed or flushed. You do not have to pass a preallocated block of memory into this parameter as the memory will be allocated as needed. Upon return of the function, BufLoc will point to the stream's data.

SizeLoc: Upon return of the function, this parameter will hold the size in bytes of the data held in BufLoc.

Return Value

If successful, the function returns a valid file stream. If unsuccessful, the function returns nil. The reason the file stream could not be created will be returned by the `errno` function.

See Also

`fopen`, `fflush`, `fclose`

Example

Listing 3.68 - Using the `open_memstream` function

```
procedure TForm1.Button1Click(Sender: TObject);
var
    memStr: PIOFile;
    Buffer: PChar;
    BufferSize: integer;
begin
    //Create the Memory Stream
    memStr := open_memstream(@Buffer, @BufferSize);

    //Write into the Buffer
    fprintf(memStr, 'The next random number is %d', random(100));

    //Close the stream. This will also push the data from
    //the stream into the Buffer.
```

```

fclose(memStr);

mmLines.Lines.Add(StrPas(Buffer));
end;

```

opendir ***LibC.pas***

Syntax

```

function opendir(
  PathName: PChar
):Pointer;

```

Description

This function returns a pointer to a directory stream that can be used to obtain a list of files and directories. Once the directory stream handle has been obtained, entries can be read from the directory stream using the `readdir` function.

Parameters

PathName: The `PathName` parameter holds the location of the directory such as `/usr/someuser/`. An invalid value in this parameter will result in the function failing and a call to `errno` returning `ENOENT`. Permission to the directory should also be obtained in order to read the files.

Return Value

When successful, the function returns a valid pointer to a directory stream. If the function was unsuccessful for any reason, the function returns a nil pointer.

See Also

`readdir`, `closedir`

Example

See Listing 3.18 - Reading a list of files in a directory.

pread ***LibC.pas***

Syntax

```

function pread(
  FileDes: Integer;
  Buf: Pointer;
  NBytes: LongWord;
  Offset: Integer
):Integer;

```

Description

The `pread` function is similar to the `__read` function found in the `LibC` unit, except that it reads data into the buffer at a specific location from the file descriptor. The file descriptor's read/write position is not affected by a call to `pread`.

Parameters

FileDes: This is a file descriptor obtained by a call to `open` or another file that returns a valid file descriptor.

Buf: This is the location where the data will be read into. It should have at least `NBytes` bytes of memory.

NBytes: This is the number of bytes that should be read into the buffer.

Offset: This is the position in the file descriptor from which the file should begin reading the data.

Return Value

When successful, the function returns the number of bytes that were successfully written into the buffer. If the function was unsuccessful, it returns `-1`.

See Also

`fwrite`, `open`

putw LibC.pas**Syntax**

```
function putw(
  W: Integer;
  Stream: PIOFile
):Integer;
```

Description

The `putw` function inserts an integer value `W` into the stream. This is a backward compatibility function; a call to `fwrite` is preferred over calling this function.

Parameters

W: This is the integer value to insert into the stream.

Stream: This is the file to which the integer will be written.

Return Value

Upon success, this function returns `0`. If the function fails, it will return `-1`.

See Also

`getw`, `fwrite`, `fopen`

Example**Listing 3.69 - Using the putw function**

```
procedure TfrmMain.btnSaveValuesClick(Sender: TObject);
var
  fs: PIOFile;
begin
```

```
//Make sure a filename is entered
if edFilename.Text <> '' then
begin
  //Write three values to file
  fs := fopen(PChar(edFilename.Text), 'w');
  putw(SpinEdit1.Value, fs);
  putw(SpinEdit2.Value, fs);
  putw(SpinEdit3.Value, fs);
  fclose(fs);
end;
end;
```

pwrite ***LibC.pas***

Syntax

```
function pwrite(
  FileDes: Integer;
  const Buf: Pointer;
  N: LongWord;
  Offset: Integer
):Integer;
```

Description

This function will write a buffer of data into a file stream at a specific position in the file descriptor. Calling this function would be the equivalent of storing the file position, moving the file position to Offset, writing the data, and then restoring the original file position.

Parameters

FileDes: This is the file descriptor to where the data from the Buf parameter will be written.

Buf: This is the pointer to the data that will be written to the file descriptor.

N: This is the amount of bytes from the buffer Buf that will be written to the file descriptor.

Offset: This is the position of the file descriptor to which the data will be written.

Return Value

When successful, the function returns the number of bytes that were successfully written into the file descriptor. If the function was unsuccessful, it returns -1.

See Also

`pread`

readdir, readdir64 ***LibC.pas***

Syntax

```
function readdir(
  Handle: PDirectoryStream
):PDirEnt;
```



```
function readdir64(
  Handle: PDirectoryStream
):PDirEnt64;
```

Description

The `readdir` function reads a `TDirEnt` from a directory stream which was obtained by a call to `opendir`. The information that is returned by this function allows you to examine the details about the entry in the directory.

Parameters

Handle: This is a handle to a valid directory stream. This value is normally obtained by a call to the `opendir` function.

Return Value

When successful, this function returns a pointer to a `TDirEnt` structure.

```
type
  dirent = packed record
    d_ino:      LongInt;
    d_off:      Integer;
    d_reclen:   Word;
    d_type:     char;           //Type of entry
    d_name:     array [0..255] of char; //Name of the entry
  end;
  TDirEnt = dirent;
  PDirEnt = ^TDirEnt;
```

After a call to `readdir`, the `TDirEnt` structure is not complete on all Linux systems and only some information in the `TDirEnt` record is reliable such as the `d_name`, which represents the recently read entry. The `d_type` field in the `TDirEnt` structure should return a value from the constant list in Listing 3.70, but on some implementations the only value that returns is `DT_UNKNOWN`. The only information that is guaranteed to be returned in `TDirEnt` is the `d_name` field.

Listing 3.70 - Options for the `TDirEnt`'s `d_type` field

```
Const
  DT_UNKNOWN  = 0;
  DT_FIFO    = 1;
  DT_CHR     = 2;
  DT_DIR     = 4;
  DT_BLK     = 6;
  DT_REG     = 8;
  DT_LNK     = 10;
  DT_SOCK    = 12;
```

Most of this information can be obtained by using any one of the `stat` functions such as `stat` or `fstat`.

When unsuccessful, the function returns `nil`. The most common problem for this function is an invalid directory stream handle being passed into the function.

See Also

opendir, closedir, telldir, rewinddir, seekdir, telldir

Example

See Listing 3.18 - Reading a list of files in a directory.

remove LibC.pas**Syntax**

```
function remove(
  FileName: Pchar
):Integer;
```

Description

This function removes a file or directory specified in the FileName parameter. In the case of a file, if it is linked and is the last linked file, and no processes are accessing the file, then it will be removed.

Parameters

FileName: This is either the name of a file or the name of a directory.

Return Value

Upon success, the function returns 0. The function will return -1 if it failed. In most cases the function will fail due to insufficient permissions to remove the file or because the file does not exist. Check the value of errno to analyze the reason for the failure.

Example**Listing 3.71 - Using the remove function**

```
//Remove a file
remove('/tmp/somefile.txt');
//Remove a directory
remove('/tmp/MyTempDir');
```

rewind LibC.pas**Syntax**

```
procedure rewind(
  Stream: PIOFile
);
```

Description

The rewind function sets the read/write position of a stream to the beginning of the stream and clears any errors. It is the equivalent of calling the following:

```
//Go to the start of the stream
fseek(Stream, 0, SEEK_SET);
//Clear any errors
clearerr(Stream);
```

Parameters

Stream: This is the stream that will be rewound.

See Also

rewinddir, fseek, clearerr

Return Value

The rewind function never returns a value.

Example

Listing 3.72 - Using the rewind function

```
var
  dBaseFile: PIOFile;
  FileHeader: TdBaseHeader;

procedure OpenDBaseFile;
begin
  //Read in the header
  dBaseFile := fopen(PChar(Filename), 'r');
  fread(@FFileHeader, sizeof(TdBaseHeader), 1, dBaseFile);
  //Go back to the start
  rewind(dBaseFile);
end;
```

rewinddir LibC.pas

Syntax

```
procedure rewinddir(
  Handle: PDirectoryStream
);
```

Description

The function sets the read position of a directory stream created with opendir to the start of the directory stream. At the same time, the function will pick up any new files or directories that have been added to or removed from the directory since the initial call to opendir. This function can also be used to refresh a directory stream's information.

Parameters

Handle: This is the directory stream that will be reinitialized.

See Also

opendir, closedir, readdir

*Example***Listing 3.73 - Using the rewinddir function**

```

procedure TForm1.FormCreate(Sender: TObject);
var
  dirName: array[0..500] of char;
begin
  getcwd(dirName, 500);
  FCurrentDir := dirName;

  //Open the directory stream
  DirStream := opendir(dirName);

  //Display the directory information
  ReadInDirectory;
end;

procedure TForm1.ReadInDirectory;
var
  item: TListItem;
  PDirInfo: PDirEnt;
begin
  //Rewind the Directory
  rewinddir(DirStream);

  //Clear the List View
  lvFiles.Items.Clear;

  //Here is where we read the entries of the directory
  if DirStream <> nil then
  begin
    PDirInfo := readdir(DirStream);
    while PDirInfo <> nil do
    begin
      item := lvFiles.Items.Add;
      item.Caption := PDirInfo^.d_name;
      PDirInfo := readdir(DirStream);
    end;
  end;
end;

```

scandir, scandir64 LibC.pas***Syntax***

```

function scandir(
  PathName: PChar;
  var NameList: PPDirent;
  SelProc: TSelectorProc;
  CmpProc: TCompareProc
):Integer;

```

```

function scandir64(
  PathName: PChar;

```

```

var NameList: PPDirent64;
SelProc: TSelectorProc64
): Integer;

```

Description

This function returns a selection of entries within a directory and allows ways to filter and sort the entries that are returned.

Parameters

PathName: This is the name of the directory where the directory entries will be listed.

NameList: This value returns a list of pointers containing the entries of the directory that are returned.

SelProc: This is a pointer to a TSelectorProc function that defines whether a specific directory entry should be returned in the NameList parameter. If this parameter is nil, then all directory entries will be removed from the list.

```
TSelectorProc = function(const p1: PDirEnt): Integer; cdecl;
```

```
//An example TSelectorProc
```

```

function IsValidDirectory(const p1: PDirEnt): Integer; cdecl;
var
    dirName: string;
begin
    //Here we filter to see what directory entries are acceptable.
    dirName := p1^.d_name;
    if (dirName = '.') or (dirName = '..') then
        Result := 0
    else
        Result := 1;
end;

```

CmpProc: This is the comparison function that will be used against the list of directory entries. The value that is normally passed here is a call to `alphasort` or `versionsort`. A custom written function can be used here also, to allow functionality such as sorting by dates. If nil is passed as a parameter to `CmpProc`, the returned list is not sorted.

Return Value

The function returns the number of entries that can be obtained from the NameList parameter when successful or -1 if unsuccessful.

See Also

`alphasort`, `versionsort`, `opendir`, `readdir`, `closedir`

Example

Listing 3.74 - Using the scandir function

```

function IsValidDirectory(const p1: PDirEnt): Integer; cdecl;
var
    dirName: string;

```

```

begin
    //Here we filter to see what directory entries are acceptable.
    dirName := p1^.d_name;
    if (dirname = '..') or (dirname = '.') then
        Result := 0
    else
        Result := 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    dirCount: integer;
    counter: integer;
    entries: PPDirent;
    CurrentEntry: TDirEnt;
begin
    //Clear the Items
    lbEntries.Items.Clear;

    //Get the Entries using a sorting algorithm.
    if cbNoSorting.Checked then
        dirCount := scandir(PChar(edDirectory.Text), entries, IsValidDirectory, nil)
    else if cbUseAlphaSort.Checked then
        dirCount := scandir(PChar(edDirectory.Text), entries, IsValidDirectory, alphasort)
    else if cbUseVersionSort.Checked then
        dirCount := scandir(PChar(edDirectory.Text), entries, IsValidDirectory,
                             versionsort);

    //Display the results
    for counter := 0 to dirCount - 1 do
    begin
        //Add the directory entry to the ListBox.
        CurrentEntry := entries^[counter];
        lbEntries.Items.Add(CurrentEntry.d_name);

        //Forward the position
        entries := PPDirent(Integer(entries) + sizeof(PDirent));
    end;
end;

```

seekdir LibC.pas

Syntax

```

procedure seekdir(
    Handle: PDirectoryStream;
    Position: Integer
);

```

Description

The seekdir function sets the reading position of a directory stream to the value passed into the Position parameter. The Position parameter is obtained by a previous call to telldir.

Parameters

Handle: Handle is a valid directory stream that was opened with a call to `opendir`.

Position: The Position parameter sets the new position of the directory stream. This value is obtained by a call to `telldir`.

See Also

`telldir`, `opendir`, `rewinddir`

Example

Listing 3.75 - Using the `seekdir` function

```

procedure TfrmMain.ReadInDirectory;
var
  item: TListItem;
  PDirInfo: PDirEnt;
  BackDirPos: integer;
begin
  //Rewind the Directory
  rewinddir(DirStream);

  //Clear the List View
  lvFiles.Items.Clear;

  //Here is where we read the entries of the directory
  if DirStream <> nil then
  begin
    PDirInfo := readdir(DirStream);
    while PDirInfo <> nil do
    begin
      if PDirInfo^.d_name = '..' then
        BackDirPos := telldir(DirStream);
      item := lvFiles.Items.Add;
      item.Caption := PDirInfo^.d_name;
      PDirInfo := readdir(DirStream);
    end;
  end;

  //Goto to the rewind position
  seekdir(DirStream, BackDirPos);

  //Now get the first real file
  PDirInfo := readdir(DirStream);
  if PDirInfo <> nil then
    Caption := 'The first real file is: ' + PDirInfo^.d_name;
end;

```

setbuf *LibC.pas*

Syntax

```

procedure setbuf(
  Stream: PIOFile;

```

```
Buf: PChar
);
```

Description

The `setbuf` function defines whether a stream will be fully buffered, that is, content is held in memory until BUFSIZ bytes have been written to the stream before the contents are written to disk, or unbuffered, where content is written directly to disk.

Parameters

Stream: This is the stream that will have its buffering defined.

Buf: When the Buf parameter is nil, the stream will be unbuffered. Otherwise, buffering will be turned on and the size of the buffer should be BUFSIZ bytes.

See Also

`setvbuf`, `setbuffer`, `setlinebuf`

Example

Listing 3.76 - Using the `setbuf` function

```
procedure TfrmMain.btnSetBufClick(Sender: TObject);
var
  p: PChar;
begin
  if rbFullyBuffered_setbuf.Checked then
  begin
    //Set the Stream to be buffered
    p := malloc(BUFSIZ);
    setbuf(dataFile, p);
  end else if rbUnbuffered_setbuf.Checked then begin
    //Set the Stream to be unbuffered
    setbuf(dataFile, nil);
  end;
end;
```

setbuffer *LibC.pas*

Syntax

```
procedure setbuffer(
  Stream: PIOFile;
  Buf: PChar;
  Size: Integer
);
```

Description

The `setbuffer` function sets a stream to be unbuffered or fully buffered using Buf, which is Size bytes long.

Parameters

Stream: This is the stream that will have its buffering level defined.

Buf: This is either a pointer to where the buffering will be held, in which case the stream will be fully buffered or nil, indicating the stream will be unbuffered.

Size: When the Buf parameter is not nil, the Size parameter defines the size of the buffer.

See Also

setbuf, setvbuf, setlinebuf

Example**Listing 3.77 - Using the setbuffer function**

```
procedure TfrmMain.btnSetBufferClick(Sender: TObject);
var
  p: PChar;
begin
  if rbFullyBuffered_setbuffer.Checked then
  begin
    //Set the stream to be buffered
    p := malloc(strtoint(edBufferSize_setbuffer.Text));
    setbuffer(dataFile, p, strtoint(edBufferSize_setbuffer.Text));
  end else begin
    //Set the stream to be unbuffered
    setbuffer(datafile, nil, 0);
  end;
end;
```

setlinebuf LibC.pas**Syntax**

```
procedure setlinebuf(
  Stream: PIOFile
);
```

Description

The setlinebuf function sets a stream to be line buffered. Only when a new line is written to the stream will the write operations be written to disk. All the memory to set up the buffering is allocated from this function.

Parameters

Stream: This is the stream that is to have its buffering set to line buffering.

See Also

setbuf, setbuffer, setvbuf

Example

Listing 3.78 - Using the setlinebuf function

```

procedure TfrmMain.btnSetLineBufClick(Sender: TObject);
begin
    //Set the buffer to use Line buffering.
    setlinebuf(dataFile);
end;

```

setvbuf LibC.pas

Syntax

```

function setvbuf(
    Stream: PIOFile;
    Buf: PChar;
    Modes: Integer;
    N: LongWord
):Integer;

```

Description

This function determines the method of data buffering that is used for a stream. Although your application may be writing individual characters to a file, the data may not be written until a block of data can be written, depending on the buffering method. There are three valid options for buffering. The first option is full buffering. With full buffering, data is written into a data buffer; when the buffer is full, the contents of the buffer are written to disk. Another option is line buffering, where data is written to disk only when a new line is written. The final option that is available is no buffering, in which case data is immediately written to disk.

Parameters

Stream: This is the stream that will have its buffering mode set.

Buf: When using full buffering or line buffering, this is where the data will be stored when it is sent to disk. It is important that this buffer exist in memory whenever a call to the stream is made, so do not have the buffer set to a local variable unless all operations on the stream are done within the local function. Setting this parameter to nil causes the memory to be allocated by the function.

Modes: This is the buffering mode that will be used on the stream and is one of the following values:

- `_IOFBF` — Full buffering is used on the stream.
- `_IOLBF` — Line buffering is used on the stream.
- `_IONBF` — No buffering is used on the stream.

N: This is the size of the buffer in Buf. This value should either be the constant `BUFSIZ` or taken from a `TStatBuf`'s `st_blksize` field after an `fstat` function has been called on the file stream.

Return Value

When successful, the function returns 0. If the function fails, the result is -1.

See Also

fstat, stat, setlinebuf, setbuf, setbuffer

Example

Listing 3.79 - Using the setvbuf function

```
procedure TfrmMain.btnSetvBufClick(Sender: TObject);
var
  p: PChar;
begin
  if rbFullyBuffered_setvbuf.Checked then
    begin
      //Initialize the size of the buffer
      p := malloc(strtoint(edBufferSize_setvbuf.Text));
      //Set the stream to be fully buffered
      setvbuf(dataFile, p, _IOFBF, strtoint(edBufferSize_setvbuf.Text));
    end else if rbUnbuffered_setvbuf.Checked then begin
      setvbuf(dataFile, nil, _IONBF, 0);
    end else if rbLineBuffered_setvbuf.Checked then begin
      //Initialise the size of the buffer
      p := malloc(strtoint(edBufferSize_setvbuf.Text));
      //Set the stream to be line buffered
      setvbuf(dataFile, p, _IOLBF, strtoint(edBufferSize_setvbuf.Text));
    end;
end;
```

stat, stat64 LibC.pas

Syntax

```
function stat(
  FileName: PChar;
  var StatBuffer: TStatBuf
):Integer;

function stat64(
  FileName: PChar;
  var StatBuffer: TStatBuf64
):Integer;
```

Description

This function returns the details about a particular file or directory, such as permissions of the file, the type of file, and ownership information. The information about the FileName file is stored in a TStatBuf as seen in Listing 3.13. Your application will only be able to retrieve the information if you have permission to list the filename's directory.

Parameters

FileName: This is the name of the file for which you wish to retrieve the information.

StatBuffer: This is a TStatBuf variable that after calling the function will be filled with the information about the file. More details on analyzing the TStatBuf record can be found in Listing 3.13.

Return Value

The stat function returns 0 when successful and -1 if it failed. An error will normally only occur when the file passed to the function does not exist.

See Also

lstat, fstat

Example

See Listing 3.18 - Reading a list of files in a directory.

telldir *LibC.pas*

Syntax

```
function telldir(  
  Handle: PDirectoryStream  
):Integer;
```

Description

The telldir function returns the current read position of a directory stream. This function is used to store the position for later calls to seekdir to set the directory to the position at the time of the telldir call.

Parameters

Handle: This is a valid directory stream from a previous call to opendir.

Return Value

The function returns an integer value that represents the current read position of the directory stream. The return value will be -1 if there was an error.

See Also

seekdir, opendir, rewinddir

Example

Listing 3.80 - Using the telldir function

```
procedure TfrmMain.ReadInDirectory;  
var  
  item: TListItem;  
  PDirInfo: PDirEnt;  
  BackDirPos: integer;
```

```

begin
    //Rewind the Directory
    rewinddir(DirStream);

    //Clear the List View
    lvFiles.Items.Clear;

    //Here is where we read the entries of the directory
    if DirStream <> nil then
    begin
        PDirInfo := readdir(DirStream);
        while PDirInfo <> nil do
        begin
            if PDirInfo^.d_name = '..' then
                BackDirPos := telldir(DirStream);
            item := lvFiles.Items.Add;
            item.Caption := PDirInfo^.d_name;
            PDirInfo := readdir(DirStream);
        end;
    end;

    //Goto to the rewind position
    seekdir(DirStream, BackDirPos);

    //Now get the first real file
    PDirInfo := readdir(DirStream);
    if PDirInfo <> nil then
        Caption := 'The first real file is: ' + PDirInfo^.d_name;
end;

```

tempnam LibC.pas

Syntax

```

function tempnam(
    Dir: PChar;
    PFX: PChar
):PChar;

```

Description

This function returns the name of a temporary file. At the time the function was called, the filename that is returned did not exist.

Parameters

Dir: This is the directory where the temporary file is to be created. This should be a valid path and the user who is running the application should have the appropriate permissions to create and write to the file in this directory. If this value is nil, then the value from the environment variable TEMPDIR is used as the directory.

PFX: This is the suffix of the temporary file that is to be created.

Return Value

When successful, this function will return a valid filename; otherwise, the function will return nil. When the function is unsuccessful, a call to `errno` should return the error. The most likely problems to occur with this function is that the directory parameter is invalid or the user does not have the appropriate permissions to work with this directory.

See Also

`tmpfile`, `tmpfile64`

Example

Listing 3.81 - Using the `tempnam` function

```
function GetTempFileName: string;
var
    strFileName: PChar;
begin
    strFilename := tempnam(nil, 'tmp');
    Result := StrPas(strFilename);
end;
```

`tmpfile`, `tmpfile64` LibC.pas

Syntax

```
function tmpfile:PIOFile;
```

```
function tmpfile64:PIOFile;
```

Description

The `tmpfile` function returns a newly created temporary stream for writing in binary mode equivalent to opening with a mode of `wb+`. This function is much safer than using the `tempnam` function as this function guarantees that the temporary file did not exist before it was created, unlike `tempnam` which has a chance of returning a filename that may be taken by another process.

Return Value

The function returns a valid stream when successful and nil if it failed.

Example

Listing 3.82 - Using the `tmpfile` function

```
var
    dataFile: PIOFile;

procedure TfrmMain.btnCreateTmpFileClick(Sender: TObject);
begin
    //Create a temporary file that we can use
    dataFile := tmpfile;
end;
```

tmpnam, tmpnam_r LibC.pas**Syntax**

```
function tmpnam(
  S: PChar
):PChar;

function tmpnam_r(
  S: PChar
):PChar;
```

Description

This function returns the name of a file that, at the time of calling this function, does not exist on the file system. Although this function returns a nonexistent filename, other processes and threads may call this function and obtain the same filename, so you cannot be guaranteed that the filename returned is usable. A safer way to obtain a file is to use the `tmpfile` function.

Parameters

S: This is a nil pointer that means the resulting value after the function call will point to a LibC internal string. If this parameter is allocated memory, the buffer that this PChar points to will be filled with a new filename.

Return Value

The function returns S if successful and nil if unsuccessful.

See Also

`tmpfile`, `tempnam`

Example**Listing 3.83 - Using the tmpnam function**

```
procedure TForm1.Button1Click(Sender: TObject);
var
  p1: array[0..L_tmpnam] of char;
begin
  if tmpnam(p1) <> nil then
    edTemporaryFilename.Text := StrPas(p1);
end;
```

ungetc LibC.pas**Syntax**

```
function ungetc(
  C: Integer;
  Stream: PIOFile
):Integer
```

Description

This function will push back a character *C* into the stream so that it is the next character read. If the stream is at the end of file (EOF), calling this function will clear the end of file flag on the stream.

Parameters

C: This is the character that is to be pushed back into the stream, typecast as an integer. If you wish to use a single character, you should set *C* to `ord(someCharacter)`. This character does not have to be the last file that was read in from the stream.

Stream: This is the stream where the file will be pushed back into.

Return Value

Upon success, this function will return 0. If the function is unsuccessful, it will return `-1`. The most common reason for failure of the function is that the stream being passed into the *Stream* parameter is invalid.

See Also

`getc`, `fgetc`

Example

Listing 3.84 - Using the `ungetc` function

```
function TfrmMain.ParseInteger(Stream: PIOFile): integer;
var
  c: integer;
  sNum: string;
begin
  //This function returns a number returned from a stream
  sNum := '';

  c := getc(Stream);
  while isdigit(c) <> 0 do
  begin
    sNum := sNum + chr(c);
    c := getc(Stream);
  end;

  //Re-insert the last character read as it was not
  //part of the number
  ungetc(c, Stream);

  //Return the Result
  Result := strtointDef(sNum, -1);
End;
```


versionsort, versionsort64 LibC.pas**Syntax**

```
function versionsort(
  const e1: Pointer;
  const e2: Pointer
):Integer;

function versionsort64(
  const e1: Pointer;
  const e2: Pointer
):Integer;
```

The versionsort and versionsort64 functions are built-in sorting routines designed to arrange a list of directory entries in alphabetical order. The function is normally passed to a function such as scandir so that it can be used when sorting the entries of a directory.

Parameters

- e1: This is a pointer to a TDirEnt record used for comparing directories.
- e2: This is also a pointer to a TDirEnt record used for comparing directories.

Return Value

The function returns the value of a string comparison which is internally a call to the strverscmp function.

See Also

scandir, alphasort, alphasort64

Example**Listing 3.85 - Using the versionsort function**

```
var
  dirCount: integer;
  entries: PDirEnt;
begin
  //Use the Version Sort
  dirCount := scandir(PChar(edDirectory.Text), entries, nil, versionsort);
end;
```

Practical Example — A dBase III File Reader

The practical application for this chapter is a simple dBase file reader. Descending from TClientDataSet, this component will allow you to read in a simple dBase file that does not have memo files or indexes.

The component itself uses the Linux API when dealing with the dBase file. The dBase file is made up of several blocks of data that can be read. If you examine the source code of the components, you will see the TdBaseHeader that contains information about the dBase file, and the TdBaseInternalField record.

The component actually uses a stream to open the file and read the data. The majority of the work with streams is done with the `LoadFromdBaseFile` and `SaveTodBaseFile` methods. These methods fully demonstrate effective use of streams when working with the Linux API.

Listing 3.86 - Source code for the dBase file reader component

```
unit DBaseFileReader;

interface

uses
    SysUtils, Types, Classes, QGraphics, QDialogs, DB, DBClient, LibC;

type
    //Structure that holds the information for the dBase file
    TdBaseHeader = packed record
        InfoByte: byte;
        LastUpdateYear: byte;
        LastUpdateMonth: byte;
        LastUpdateDay: byte;
        NumberOfRecords: dword;
        BytesInTheHeader: word;
        BytesInTheRecord: word;
        Reserved1: word;
        InTransaction: byte;
        EncryptionFlag: byte;
        Reserved2: array[0..11] of char;
        HasMDXFile: byte;
        LanguageDriver: byte;
        Reserved3: word;
    end;

    //Structure that holds the information for each field.
    TdBaseInternalField = packed record
        FieldName: array[0..10] of char;
        FieldType: char;
        FieldDataAddress: dword;
        FieldLengthBinary: byte;
        FieldDecimalLengthBinary: byte;
        Reserved1: word;
        WorkAreaID: byte;
        Reserved2: word;
        SetFieldsFlag: byte;
        Reserved3: array[0..6] of char;
        HasIndex: byte;
    end;

    TdBaseFileReader = class;

    TdBaseFileFieldType = (dbaseString, dbaseNumber, dbaseDate,
        dbaseLogical, dbaseMemo, dbaseUnknown);
    TdBaseFileField = class
    private
        FField: TdBaseInternalField;
        FOwner: TdBaseFileReader;
```

```

    function GetFieldName: string;
    function GetFieldType: TdBaseFileFieldType;
public
    constructor Create(AOwner: TdBaseFileReader;
        AField: TdBaseInternalField);
    property FieldName: string read GetFieldName;
    property FieldType: TdBaseFileFieldType read GetFieldType;
end;

EdBaseFileError = class(Exception);

TdBaseFileReader = class(TClientDataSet)
private
    FFileHeader: TdBaseHeader;
    FdBaseFieldList: TList;
    function GetdBaseField(Index: integer): TdBaseFileField;
    procedure ClearBaseFields;
    function GetBlankString(intSize: integer): string;
    //Functions for converting dBase fields into Kylix field type values
    function GetdBaseString(Value: string; var ValueIsNull: boolean): string;
    function GetdBaseNumber(Value: string; declen: integer; var ValueIsNull:
        boolean): double;
    function GetdBaseDate(Value: string; var ValueIsNull: boolean): TDateTime;
    function GetdBaseLogical(Value: string; var ValueIsNull: boolean): boolean;
    //Functions for writing Kylix field values as dBase formats
    function StringTodBaseString(Value: string; FieldSize: integer;
        WriteNull: boolean): string;
    function NumberTodBaseNumber(Value: double; Prec, DecPrec: integer;
        WriteNull: boolean): string;
    function DateTodBaseDate(Value: TDateTime; WriteNull: boolean): string;
    function LogicalTodBaseLogical(Value: boolean; WriteNull: boolean): string;
protected
    dBaseFile: PIOFile;
    FFieldCount: integer;
    AtBOF, AtEOF: boolean;
    DataStartPosition: integer;
    FIsDeleted: boolean;
    RecordStartPosition: integer;
    RecordBuffer: Pointer;
public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    //Field Information functions
    function dBaseFieldCount: integer;
    property dBaseFields[Index: integer]: TdBaseFileField read GetdBaseField;
    //Main method for loading and saving the dBase file.
    procedure LoadFromdBaseFile(strFilename: string);
    procedure SaveTodBaseFile(strFilename: string);
end;

implementation

{ TdBaseFileReader }

procedure TdBaseFileReader.ClearBaseFields;
var
    counter: integer;

```

```

begin
  for counter := dBaseFieldCount - 1 downto 0 do
    begin
      //Remove the field list
      dBaseFields[counter].Free;
      FdBaseFieldList.Delete(counter);
    end;
  end;

constructor TdBaseFileReader.Create(AOwner: TComponent);
begin
  inherited;

  FdBaseFieldList := TList.Create;
end;

destructor TdBaseFileReader.Destroy;
begin
  CleardBaseFields;
  FdBaseFieldList.Free;

  inherited;
end;

function TdBaseFileReader.dBaseFieldCount: integer;
begin
  Result := FdBaseFieldList.Count;
end;

function TdBaseFileReader.GetdBaseField(Index: integer): TdBaseFileField;
begin
  Result := TdBaseFileField(FdBaseFieldList.Items[Index]);
end;

procedure TdBaseFileReader.LoadFromdBaseFile(strFilename: string);
var
  fFieldInfo: TdBaseInternalField;
  counter: integer;
  objField: TdBaseFileField;
  DataInc: integer;
  i: integer;
  RecordValue: string;
  RecordDeleted: boolean;
  CurrentOffset: integer;
  DatafieldDef: TFieldDef;
  IsNullValue: boolean;
  resString: string;
  resDate: TDateTime;
  resBoolean: boolean;
  resNumber: double;
  CurrentFieldString: string;
  CurrentFieldName: string;
begin
  //Open the dBase File
  if not FileExists(strFilename) then
    raise EdBaseFileError.Create('The dBase File you are trying to '+
      'open does not exist.');
```

```

//Clear the FieldInformation from the Dataset if it is open
Active := false;
FieldDefs.Clear;

//Open the file
dBaseFile := fopen(PChar(strFilename), 'r');

//Read in the header
fread(@FFileHeader, sizeof(FFileHeader), 1, dBaseFile);

//Allocate Memory for the Recods
RecordBuffer := AllocMem(FFileHeader.BytesInTheRecord);

//Calculate the number of fields in the dBase file
FFieldCount := (FFileHeader.BytesInTheHeader - 34) div 32;

//read the fields in
for counter := 1 to FFieldCount do
begin
    fread(@fFieldInfo, sizeof(TdBaseInternalField), 1, dBaseFile);
    fFieldInfo.FieldDataAddress := DataInc;
    objField := TdBaseFileField.Create(Self, fFieldInfo);
    fdBaseFieldList.Add(objField);

    //Add the field information to the FieldsDefs
    DatafieldDef := FieldDefs.AddFieldDef;
    DatafieldDef.Name := fFieldInfo.FieldName;
    DatafieldDef.DisplayName := fFieldInfo.FieldName;

    case fFieldInfo.FieldType of
        'C': begin
            DataFieldDef.DataType := ftString;
            DatafieldDef.Size := fFieldInfo.FieldLengthBinary;
        end;
        'N': begin
            DataFieldDef.DataType := ftFloat;
            DataFieldDef.Precision := fFieldInfo.FieldDecimalLengthBinary;
        end;
        'D': DataFieldDef.DataType := ftDate;
        'L': DataFieldDef.DataType := ftBoolean;
    end;
end;

//Confirm the field structure and open the Table
CreateDataSet;
DisableControls;
Active := true;

//Go to the start of where the data is
fseek(dBaseFile, FFileHeader.BytesInTheHeader, SEEK_SET);

//Read in all the records
for i := 1 to FFileHeader.NumberOfRecords do
begin
    //Read the record into the database.
    fread(RecordBuffer, FFileHeader.BytesInTheRecord, 1, dBaseFile);
    RecordValue := StrPas(PChar(RecordBuffer));
end;

```

```

//Read if it is deleted or not.
RecordDeleted := Copy(RecordValue, 1, 1) = '*';
if RecordDeleted then
    continue;

//Set the new offset due to the record-delete marker
CurrentOffset := 2;

//Add the record contents
Append;

for counter := 0 to FFieldCount - 1 do
begin
    CurrentFieldName := dBaseFields[counter].FieldName;
    CurrentFieldString := Copy(RecordValue, CurrentOffset,
        dBaseFields[counter].FField.FieldLengthBinary +
        dBaseFields[counter].FField.FieldDecimalLengthBinary);

    case dBaseFields[counter].FField.FieldType of
        'C': begin
            resString := GetdBaseString(CurrentFieldString, IsNullValue);
            if not IsNullValue then
                FieldByName(CurrentFieldName).AsString := resString;
            end;
        'N': begin
            resNumber := GetdBaseNumber(CurrentFieldString, dBaseFields[counter]
                .FField.FieldLengthBinary,
                IsNullValue);
            if not IsNullValue then
                FieldByName(CurrentFieldName).AsFloat := resNumber;
            end;
        'D': begin
            resDate := GetdBaseDate(CurrentFieldString, IsNullValue);
            if not IsNullValue then
                FieldByName(CurrentFieldName).AsDateTime := resDate;
            end;
        'L': begin
            resBoolean := GetdBaseLogical(CurrentFieldString, IsNullValue);
            if not IsNullValue then
                FieldByName(CurrentFieldName).AsBoolean := resBoolean;
            end;
        end;
    inc(CurrentOffset,
        dBaseFields[counter].FField.FieldLengthBinary + dBaseFields[counter]
            .FField.FieldDecimalLengthBinary);
end;

//Save the changes to the record
Post;
end;

EnableControls;

//Release the memory for the record buffer
FreeMem(RecordBuffer);

//Close the File Stream

```

```

    fclose(dBaseFile);
end;

function TdBaseFileReader.GetdBaseDate(Value: string; var ValueIsNull: boolean):
TDateTime;
var
    wYear, wMonth, wDay: word;
begin
    try
        if Trim(Value) = '' then
            begin
                ValueIsNull := true;
            end else begin
                wYear := strtoint(Copy(Value, 1, 4));
                wMonth := strtoint(Copy(Value, 5, 2));
                wDay := strtoint(Copy(Value, 7, 2));

                Result := EncodeDate(wYear, wMonth, wDay);
                ValueIsNull := false;
            end;
        except
            ValueIsNull := true;
        end;
    end;
end;

function TdBaseFileReader.GetdBaseLogical(Value: string; var ValueIsNull: boolean):
boolean;
begin
    if Length(Value) = 0 then
        Result := false
    else begin
        Result := Value[1] in ['t', 'T', 'y', 'Y'];
        //Question Marks for Logical Fields denotes a null value
        ValueIsNull := Value[1] = '?';
    end;
end;

function TdBaseFileReader.GetdBaseNumber(Value: string; decLen: integer; var
ValueIsNull: boolean): double;
var
    strFloat: string;
begin
    try
        if Trim(Value) = '' then
            begin
                ValueIsNull := true;
            end else begin
                strFloat := Trim(Copy(Value, 1, decLen)) + '.' + Trim(Copy(Value, decLen + 1,
                    Length(Value)));
                Result := StrToFloat(strFloat);
                ValueIsNull := false;
            end;
        except
            ValueIsNull := true;
        end;
    end;
end;

```

```

function TdBaseFileReader.GetdBaseString(Value: string; var ValueIsNull: boolean):
string;
begin
    Result := Value;
    ValueIsNull := Trim(Value) = '';
end;

{ TdBaseFileField }

constructor TdBaseFileField.Create(AOwner: TdBaseFileReader;
    AField: TdBaseInternalField);
begin
    inherited Create;

    FOwner := AOwner;
    FField := AField;
end;

function TdBaseFileField.GetFieldName: string;
begin
    Result := FField.FieldName;
end;

function TdBaseFileField.GetFieldType: TdBaseFileFieldType;
begin
    case FField.FieldType of
        'C': Result := dbaseString;
        'N': Result := dbaseNumber;
        'D': Result := dbaseDate;
        'L': Result := dbaseLogical;
        'M': Result := dbaseMemo;
    else
        Result := dbaseUnknown;
    end;
end;

procedure TdBaseFileReader.SaveTodBaseFile(strFilename: string);
var
    counter: integer;
    EmptyBytesToWrite: integer;
    RecordString: string;
    CurrentField: TdBaseInternalField;
begin
    //Make sure that the structure has been previously
    //created with LoadFromdBaseFile
    if Fields.Count = 0 then
        raise EdBaseFileError.Create('The file cannot be saved as its structure was not '+
            'loaded from a dBase file.');
```

//Write the dBase file out to disk.

```

    if FileExists(strFilename) then
        unlink(PChar(strFilename));

    dBaseFile := fopen(PChar(strFilename), 'w+');
```



```

//Update the number of records
FFileHeader.NumberOfRecords := RecordCount;

//Write the contents of the dBase file out to disk.
//Write the Header
fwrite(@FFileHeader, sizeof(TdBaseHeader), 1, dBaseFile);

//Write the field information
for counter := 0 to dBaseFieldCount - 1 do
    fwrite(@(dBaseFields[counter].FField), sizeof(TdBaseInternalField), 1, dBaseFile);

//Write the space that is left
EmptyBytesToWrite := FFileHeader.BytesInTheHeader -
    (sizeof(TdBaseHeader) + dBaseFieldCount * sizeof(TdBaseInternalField)) - 1;

for counter := 1 to EmptyBytesToWrite do
    fputc(ord(' '), dBaseFile);

//Write the End of Header Marker
fputc($0D, dBaseFile);

DisableControls;

//Create the Buffer for the Record and its appropriate size.
RecordBuffer := AllocMem(FFileHeader.BytesInTheRecord);

//Write the Records.
First;
while not Eof do
begin
    //Write the individual record out.

    //Write the fact that the record is not deleted
    RecordString := ' ';

    //Write out the Values of the fields
    for counter := 0 to dBaseFieldCount - 1 do
    begin
        CurrentField := dBaseFields[counter].FField;
        case CurrentField.FieldType of
            'C': begin
                if FieldByName(CurrentField.FieldName).IsNull then
                    RecordString := RecordString +
                        StringTodBaseString('', CurrentField.FieldLengthBinary, true)
                else
                    RecordString := RecordString +
                        StringTodBaseString(FieldByName(CurrentField.FieldName).AsString,
                            CurrentField.FieldLengthBinary, false);
            end;
            'N': begin
                if FieldByName(CurrentField.FieldName).IsNull then
                    RecordString := RecordString +
                        NumberTodBaseNumber(0, CurrentField.FieldLengthBinary,
                            CurrentField.FieldDecimalLengthBinary, true)
                else
                    RecordString := RecordString +
                        NumberTodBaseNumber(FieldByName(CurrentField.FieldName).AsFloat,

```

```

        CurrentField.FieldLengthBinary, CurrentField.FieldDecimalLength-
        Binary, false);
    end;
'D': begin
    if FieldByName(CurrentField.FieldName).IsNull then
        RecordString := RecordString + DateTodBaseDate(0, true)
    else
        RecordString := RecordString +
            DateTodBaseDate(FieldByName(CurrentField.FieldName).AsdateTime,
            false);
    end;
'L': begin
    if FieldByName(CurrentField.FieldName).IsNull then
        RecordString := RecordString +
            LogicalTodBaseLogical(false, true)
    else
        RecordString := RecordString +
            LogicalTodBaseLogical(FieldByName(CurrentField.FieldName)
            .AsBoolean, false);
    end;
end;
end;

StrPCopy(RecordBuffer, RecordString);
//Write the record to the file
fwrite(RecordBuffer, FFileHeader.BytesInTheRecord, 1, dBaseFile);

Next;
end;

EnableControls;

//Close the file
fclose(dBaseFile);
end;

function TdBaseFileReader.DateTodBaseDate(Value: TDateTime;
WriteNull: boolean): string;
begin
    if WriteNull then
        Result := ' '
    else
        Result := FormatDateTime('YYYYMMDD', Value);
end;

function TdBaseFileReader.LogicalTodBaseLogical(Value,
WriteNull: boolean): string;
begin
    if WriteNull then
        Result := '?'
    else begin
        if Value then
            Result := 'Y'
        else
            Result := 'N';
    end;
end;
end;

```

```

function TdBaseFileReader.NumberTodBaseNumber(Value: double; Prec,
    DecPrec: integer; WriteNull: boolean): string;
begin
    if WriteNull then
        Result := GetBlankString(Prec + DecPrec)
    else begin
        //Write the formatted number out
        Result := FloatToStrF(Value, ffNumber, Prec, DecPrec);
        if Length(Result) < (Prec + DecPrec) then
            Result := GetBlankString((Prec + DecPrec)
                - Length(Result)) + Result;
    end;
end;

function TdBaseFileReader.StringTodBaseString(Value: string;
    FieldSize: integer; WriteNull: boolean): string;
begin
    if WriteNull then
        Result := GetBlankString(FieldSize)
    else begin
        Result := Copy(Value, 1, FieldSize);
        if Length(Result) < FieldSize then
            Result := Result + GetBlankString(FieldSize - Length(Result));
    end;
end;

function TdBaseFileReader.GetBlankString(intSize: integer): string;
var
    i: integer;
begin
    Result := '';
    for i := 1 to intSize do
        Result := Result + ' ';
    end;
end.

```

Where to Go from Here

This chapter has covered everything that you would likely use files for under Linux. From file descriptors to streams, there are ways of implementing almost any file operation that you can think of. But there is much more you can do with files under Linux. As you will see in other chapters, you can use files as a means of communicating between applications.

Before we look at communicating between processes, let's examine the fundamental aspect of every Linux application — the process.

Processes

Introduction

Processes are the lifeblood of any Linux application. A process is a unique set of memory, operational instructions, environment space, and at least a single thread of execution to make a unique context in which an application can run.

In this chapter you will examine the dealings of a process from within Linux. You will learn how processes are created under Linux and how to obtain information about processes. You will also examine other concepts that deal with processes, such as environment variables, process groups, and executing other applications. In addition you will learn techniques to examine the users and groups of the Linux system, “fork”ing the process to create an exact copy of the process, and creating silent processes called daemons that can run in the background forever.

But for now, let’s start small and look at the fundamental element for this chapter — the process.

What is a Process?

The first question that you may be asking yourself is what is a process? A process consists of a single memory space with instructions, memory, current directory information, resource usage information, user and group permissions, environment space, and at least one thread to execute code.

Each process under Linux is allocated CPU time and is also allocated a unique identifier for the process known as the process ID. The process ID is always a positive integer. Each process can have a number of child processes and a parent process that created the process.

Process ID
Private Memory
User and Group Information
Environment Values
Path Information
Resource Information
Executable Code

Figure 4.1 - What’s in a process

Gathering Information about a Process under Linux

Every process that is created under Linux has a process identifier associated with it, and information about each process identifier is stored internally within Linux. If you use the Linux `ps` function as shown in Figure 4.2, you can observe what processes are currently on the Linux system, such as what process created the process, how long it has been active, and how much CPU time the process has actually received.

Figure 4.2 -
Looking at the
processes on
the system

```

root@localhost.localdomain: /root
File Sessions Options Help

[root@localhost /root]# ps -aux | more
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0  1312    76 ?        S    22:41   0:05 init [3]
root         2  0.0  0.0      0     0 ?        SW   22:41   0:00 [kflushd]
root         3  0.0  0.0      0     0 ?        SW   22:41   0:00 [kupdate]
root         4  0.0  0.0      0     0 ?        SW   22:41   0:00 [kpiod]
root         5  0.0  0.0      0     0 ?        SW   22:41   0:01 [kswapd]
root         6  0.0  0.0      0     0 ?        SW<  22:41   0:00 [ndrecovryd]
root        61  0.0  0.0      0     0 ?        SW   22:42   0:00 [khubd]
root       277  0.0  0.1  1372   160 ?        S    22:42   0:00 syslogd -n 0
root       287  0.0  0.1  1636   176 ?        S    22:42   0:00 klogd
rpc        302  0.0  0.0   1456     0 ?        SW   22:42   0:00 [portmap]
root       318  0.0  0.0      0     0 ?        SW   22:42   0:00 [lockd]
root       319  0.0  0.0      0     0 ?        SW   22:42   0:00 [rpciod]
rpcuser    329  0.0  0.0   1552     0 ?        SW   22:42   0:00 [rpc.statd]
root       344  0.0  0.0   1296     0 ?        SW   22:42   0:00 [apmd]
nobody     398  0.0  0.0   7596    24 ?        S    22:42   0:00 [identd]
nobody     402  0.0  0.0   7596    24 ?        S    22:42   0:00 [identd]
nobody     403  0.0  0.0   7596    24 ?        S    22:42   0:00 [identd]
nobody     404  0.0  0.0   7596    24 ?        S    22:42   0:00 [identd]
nobody     405  0.0  0.0   7596    24 ?        S    22:42   0:00 [identd]
--More--

```

By looking at the processes you can gather information about which processes are running, which user is running a particular process, how long the process has been running, and what resources the process is currently using.

The PID entry in Figure 4.2 shows the unique identifier for the process; each application has one. Due to the multi-user nature of Linux you will also notice that each process is running as a particular user account. While the `ps` command lists many details about the process, you would normally only use the process identifier and user account information. Like a process is allocated a process ID (PID), each user on the Linux system is allocated an identifier that identifies which user is running the application.

In your Kylix application, obtaining the process ID and the user ID for a running process is achieved by using the `getpid` and `getuid` functions as shown in Listing 4.1.

One of the key points about working with processes is that every process has its own unique process ID. Likewise, every process is run by a particular user on the operating system. There are no exceptions to this rule.

Just as a process has access to its process ID, each process also has access to the process ID of its creator process, called its parent process ID. The parent process ID can be obtained by calling the `getppid` function. When Linux first starts up, a process is created by the operating system called the `init` process which has a process ID of 1. It is this process that creates the other processes that run the operating system.

In Listing 4.1, the application uses the `getpid` function to obtain the process identifier and the `getuid` function to obtain the number that represents the user. The `getpwuid`

function is then used to obtain display friendly information about a particular user account in order to display the information about the application.

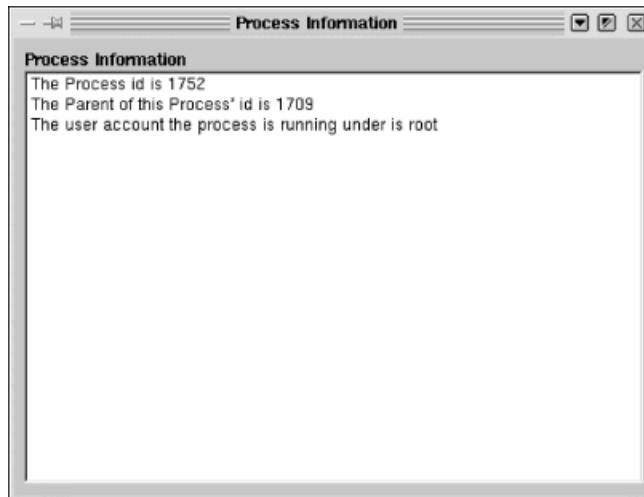
For more information about converting the user identifier to a valid user name, look at Listing 4.14.

Listing 4.1 - Displaying process information

```
function GetUserIDName(id: integer): string;
var
  pwInfo: PPasswordRecord;
begin
  pwInfo := getpwuid(id);
  Result := pwInfo^.pw_name;
end;

procedure TfrmProcessInfo.DisplaySimpleProcessInformation;
begin
  mmProcessInfo.Lines.Clear;
  mmProcessInfo.Lines.Add('The Process id is '+inttostr(getpid));
  mmProcessInfo.Lines.Add('The Parent of this Process' id is '+inttostr(getppid));
  mmProcessInfo.Lines.Add('The user account the process is running under is
    '+GetUserIDName(getuid));
end;
```

Figure 4.3 -
Looking at the
process
information

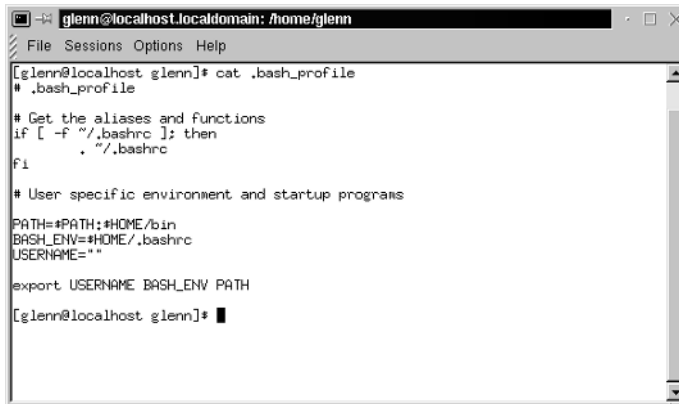


In Chapter 3 we looked at assigning user permissions to a file or directory so that particular users would have permission to access only what they needed. The permissions of the user that runs the process determines what files and directories that process has access to.

Processes and Linux Environment Variables

You saw in Figure 4.1 that each process has its own environment space. Typically, the process will obtain its environment from the parent process. In most cases, the environment will come from the shell from which the application has been run. For example, on my local Red Hat system, the environment variables come from the `.bash_profile` file located in my application's home directory. Normally, the environment will contain information about the user in the environment variables.

Figure 4.4 -
Example
environment
variables
taken from
.bash_profile



```
glenn@localhost.localdomain: ~/home/glenn
File Sessions Options Help

[glenn@localhost glenn]$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

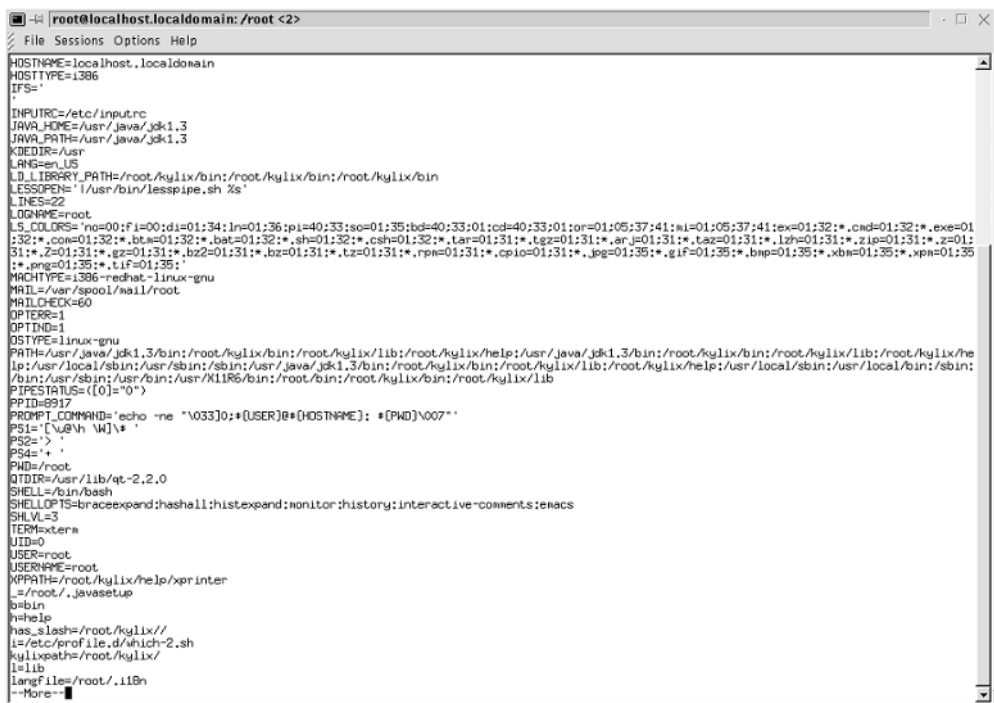
# User specific environment and startup programs

PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME=""

export USERNAME BASH_ENV PATH

[glenn@localhost glenn]$
```

Figure 4.5 -
Output from
using the set
command,
which by itself
will return the
environment
variables



```
root@localhost.localdomain: /root <2>
File Sessions Options Help

HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS='
'
INPUTRC=/etc/inputrc
JAVA_HOME=/usr/java/jdk1.3
JAVA_PATH=/usr/java/jdk1.3
KDEDIR=/usr
LANG=en_US
LD_LIBRARY_PATH=/root/kylix/bin:/root/kylix/bin:/root/kylix/bin
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=22
LOGNAME=root
LS_COLORS='no=00:fi=00:di=01:34:ln=01:36:pi=40:33:so=01:35:bd=40:33:01:cd=40:33:01:or=01:05:37:41:al=01:05:37:41:ex=01:32:*.c=01:32:*.exe=01:32:*.com=01:32:*.bat=01:32:*.sh=01:32:*.csh=01:32:*.tar=01:31:*.tgz=01:31:*.arj=01:31:*.lzh=01:31:*.zip=01:31:*.gz=01:31:*.bz2=01:31:*.bz=01:31:*.tar=01:31:*.rpm=01:31:*.cpio=01:31:*.jpg=01:35:*.gif=01:35:*.bmp=01:35:*.xbm=01:35:*.xpm=01:35:*.png=01:35:*.tif=01:35:'
MACHINE=i386-redhat-linux-gnu
MAIL=/var/spool/mail/root
MAILCHECK=60
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/usr/java/jdk1.3/bin:/root/kylix/bin:/root/kylix/lib:/root/kylix/help:/usr/java/jdk1.3/bin:/root/kylix/bin:/root/kylix/lib:/root/kylix/help:/usr/local/sbin:/usr/sbin:/sbin:/usr/java/jdk1.3/bin:/root/kylix/bin:/root/kylix/lib:/root/kylix/help:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/kylix/bin:/root/kylix/lib
PIPESTATUS=([0]=0)
PPID=9917
PROMPT_COMMAND='echo -ne "\033]0;#{USER}@#{HOSTNAME}: #{PWD}\007"'
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
PWD=/root
QTDIR=/usr/lib/qt-2.2.0
SHELL=/bin/bash
SHELLOPTS=braceexpand:hashall:histexpand:monitor:history:interactive-comments:enacs
SHLVL=3
TERM=xterm
UID=0
USER=root
USERNAME=root
XPPATH=/root/kylix/help/xprinter
_=/root/.javasetup
b=bin
h=help
has_slash=/root/kylix//
l=/etc/profile.d/which-2.sh
kylixpath=/root/kylix/
l=lib
langfile=/root/.libn
--More--
```

One of the areas in which individual user accounts differ from each other is that each individual user can define particular attributes about themselves in the form of the

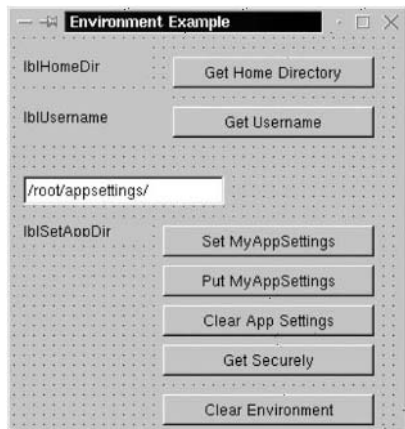
environment. Just as the real-life environment describes your surroundings, a user's environment defines the user's settings. The environment is actually a list of name/value pairs that can be accessed by individual applications, which are typically kept in one of your shell configuration files such as `.bash_profile`, shown in Figure 4.5.

Your shell will actually run your applications, and each application is responsible for passing the environment settings to their child applications. The Linux API gives you the ability to examine the environment settings through the API functions `getenv`, `setenv`, `putenv`, `clearenv`, and `__secure_env`. The first four functions are used to get, set, and clear environment values. The last function is actually used within server applications.

`__secure_env` is necessary if you are examining user values while impersonating another user or an application that runs under the guise of different user account. This is to circumvent some permission issues when certain tasks need to be performed. This method is quite common in Linux applications and is executed by setting the effective user of the application to a root user.

An Environment Example

Figure 4.6 -
A simple
environment
utility



Listing 4.2 - Source code to environment application

```
unit unitMainForm;

interface

uses
  SysUtils, Types, Classes, QGraphics, QControls,
  QForms, QDialogs, LibC, QStdCtrls;

type
  TfrmMain = class(TForm)
    Button1: TButton;
    lblHomeDir: TLabel;
    edAppSettingsDir: TEdit;
    Button2: TButton;
    lblSetAppDir: TLabel;
```



```

    lblUsername: TLabel;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Button6: TButton;
    Button7: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    frmMain: TfrmMain;

implementation

{$R *.xfm}

procedure TfrmMain.Button1Click(Sender: TObject);
begin
    //All distributions implement the HOME environment setting.
    lblHomeDir.Caption := getenv('HOME');
end;

procedure TfrmMain.Button2Click(Sender: TObject);
begin
    if setenv('appSettings', PChar(edAppSettingsDir.Text), 1) = 0 then
        lblSetAppDir.Caption := getenv('appSettings')
    else
        lblSetAppDir.Caption := 'Could not setenv.';
end;

procedure TfrmMain.Button3Click(Sender: TObject);
begin
    //All distributions implement the login name (LOGNAME)
    lblUsername.Caption := getenv('LOGNAME');
end;

procedure TfrmMain.Button4Click(Sender: TObject);
begin
    unsetenv('appSettings');
    lblSetAppDir.Caption := getenv('appSettings');
end;

procedure TfrmMain.Button5Click(Sender: TObject);
begin
    if putenv(PChar('appSettings='+edAppSettingsDir.Text)) = 0 then
        lblSetAppDir.Caption := getenv('appSettings')
    else

```

```

    lblSetAppDir.Caption := 'Could not putenv.';
end;

procedure TfrmMain.Button6Click(Sender: TObject);
begin
    //Clear the environment
    clearenv;
end;

procedure TfrmMain.Button7Click(Sender: TObject);
begin
    //We ask for the value of the environment variable
    //unless the application is running as an Superuser.
    if __secure_getenv('appSettings') <> nil then
        lblSetAppDir.Caption := __secure_getenv('appSettings')
    else
        lblSetAppDir.Caption := 'Cannot access this secure variable.';
end;

end.

```

Figure 4.6 and Listing 4.2 demonstrate a simple application that has the ability to manipulate environment variables using the `getenv`, `setenv`, `putenv`, `clearenv`, and `__secure_env` functions.

You will notice in this example that two of the environment variables that are accessed are the `HOME` and `LOGNAME` environment variables. These environment variables refer to the home directory of the current logged-on user and the logon name of the user, respectively. You can make use of these environment variables from your application by doing things such as defaulting to the `HOME` directory when using a common dialog or using the username found in the `LOGNAME` variable as a simple method for retrieving information about the current user's environment.



Note: Using the `LOGNAME` environment variables is not the recommended method for retrieving the user name because the `LOGNAME` may not be set by the shell. Instead, you should use the `getuid` API function.

These are not the only environment variables that are available. Others include `PATH`, which is used as a list of the directories that are used when searching for files. Much of the locale information about the user is also found in the environment variables, such as `LANG` for the name of the locale and `LC_MONETARY` for the definition on how to format currency values. Another popular environment variable is the `MAIL` variable. It returns where the mail directory is located. If you had an application that used e-mail for communication between remote systems, you could effectively poll this directory for any file changes to manage when new e-mail messages arrive.

Forking Processes

Recently, I went to the local cinema and saw a movie called *The 6th Day*, in which Arnold Schwarzenegger's character had a snapshot of his DNA and memory taken. It was then later cloned to create a complete duplicate of Arnold's character, memory and all.

Much like the movie where Arnold was cloned, processes under Linux can be cloned as well. The method of duplicating a process is called “forking” and is used to create a complete duplicate of a process, memory and all.

Doing this within your application is quite simple. First, the application calls the fork function and the process is copied. After the process is copied, the fork function returns twice. The function returns one value for the original process that was cloned and another value for the newly created copy of the process.

After calling fork, the original process will return the process ID of the newly created process. The newly generated process will return 0 to indicate that it is the cloned process. The code in Listing 4.3 demonstrates a simple application that forks. The resulting output from the application can be seen in Figure 4.7. If the fork function fails, the fork function will return -1 in the process that originally called it.

Listing 4.3 - A simple forking application

```
program ForkDemoApplication;

uses
  Libc;

{$APPTYPE CONSOLE}

var
  CloneProcessID: integer;

begin
  writeln(getpid, ' (Parent): Starting the Cloning Process.');
```

CloneProcessID := fork;

```

  case CloneProcessID of
    0: begin
        //We are the cloned application
        WriteLn(getpid, ' (Child): We are the Cloned Application.');
```

end;

```

    -1: begin
        //There was an error in the forking process
        writeln(getpid, ' (Parent): An error occurred while forking.');
```

end;

```

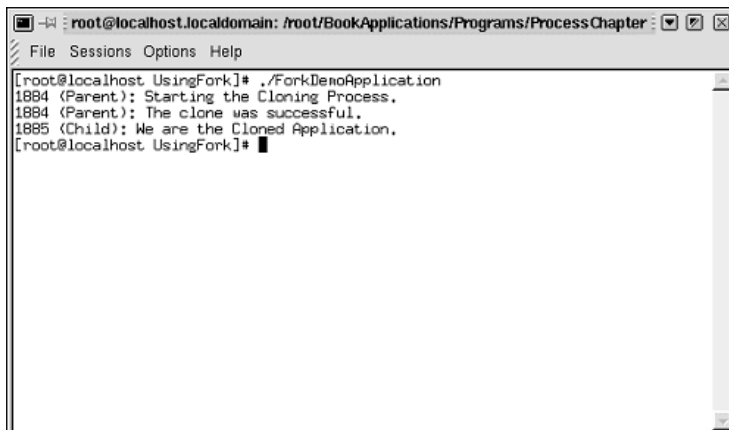
  else
    begin
        //The fork was successful
        writeln(getpid, ' (Parent): The clone was successful.');
```

end;

```

  end;
end.
```

Figure 4.7 -
Results of the
simple forking
application



```

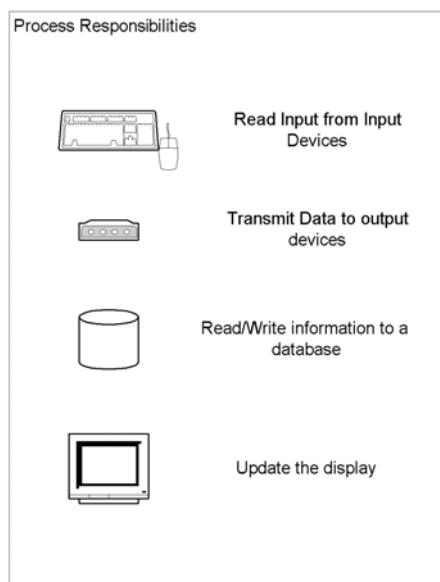
root@localhost.localdomain: /root/BookApplications/Programs/ProcessChapter
File Sessions Options Help
[root@localhost UsingFork]# ./ForkDemoApplication
1884 (Parent): Starting the Cloning Process.
1884 (Parent): The clone was successful.
1885 (Child): We are the Cloned Application.
[root@localhost UsingFork]#

```

As you can see from the simple application, what we have are two completely different processes with their own unique process identifiers. So you might be wondering, what is the practical use of forking a process? In the early days of Linux and UNIX, the ability to create separate threads did not exist, so a single individual process could only do one thing at a time.

Consider an application such as a security monitoring system that needs to obtain data from input devices, such as security systems, analyze that data, accept input from the user through the keyboard or other input device, and then update the screen to display the information. If all this functionality is contained within a single process, only one action can be performed at a time and your process may not receive some important message from a security device or it may mean that a keyboard action that performs the locking of a security gate has to wait until some other action is completed.

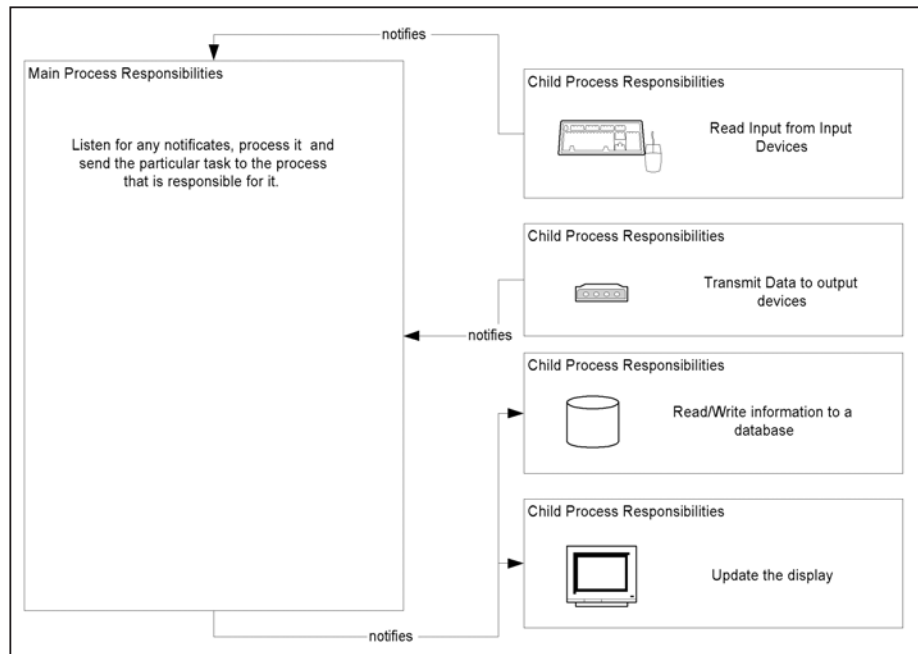
Figure 4.8 -
A single
process that
handles all
actions of the
application



This is not the ideal scenario to have when running a system that could mean the difference between life and death. So forking allows you to separate the individual components of an application so that they can all manage their particular parts of the application in separate processes. When a process has received input from whatever data source it reads from, it can communicate to the process that a change has occurred and the original process can handle the data as it is needed.

Splitting functionality between processes also adds a level of stability to an application. If the different parts of the application worked using threads, then if some code resulted in an error, the error in one thread has the potential to cause the entire application to stop. If the application was split between processes, an error in one process would not necessarily cause the whole application to close, just the problem process. In addition, the application could have code to check if required processes are running before they are communicated with. If the processes are not running, the process can be forked to create the missing process. In this way you can ensure that your application stays active longer.

Figure 4.9 - Splitting the components so that different processes handle them



Making Use of the Forked Process

As I mentioned before, when an application is forked, the entire memory space of an application is completely copied. What that means is that when you have a file that is opened in the original process, the forked process also has access to the file through the cloned file handle. Because Linux treats everything like a file, forking will also allow you to clone things like sockets, so you can implement a socket server and have the main process deal with connections. When a new connection occurs, the process forks and the child process handles the request.

When you do fork a process, you will probably want to communicate with the newly created child process in some way. You will need to look at the available interprocess communication methods that are available to you under Linux. This chapter will give you a good grounding in working with processes; however, communication between process is an entirely different area and can best be learned by looking at Chapter 5.

Process Groups

Process groups are what you think they are — groups of processes. A process group is created by an individual process that works with other processes.

For example, when a simple command on Linux, such as `cat readme.txt | more`, is executed, there are actually two processes created, the first being the process that represents the `cat` utility and the second being the process that represents the `more` utility. These processes are connected in such a way that if one process is terminated, the other process has no real reason to be around, so it should be terminated also.

When you examine process groups under the microscope, each process group has a process group ID. This process group ID is a positive integer value, just like a process ID. The process group ID will also have a process group leader. The process group leader is simply a process whose process group ID is the same integer value as the process ID.

Within your Kylix applications, process groups like this are also set up using the `setpgid` function, which will set a particular process group of a particular process. To see that the processes are connected, examine Listing 4.4, which demonstrates setting up a simple process group. Figure 4.10 also shows the application in Listing 4.4 running and shows that by killing the process group leader using the Linux `kill` command, all the processes in the process group are terminated.

You will find that apart from terminating all processes when a process group leader is terminated, there are also several functions for obtaining process group IDs, sending signals to entire process groups, and waiting for processes contained within a process group to finish executing.

Listing 4.4 - A simple demo of how process groups are connected

```
program ProcessGroupsDemo;

{$APPTYPE CONSOLE}

uses
  Libc;

var
  intChildProcessID: integer;
  counter: integer;

begin
  writeln(getpid, ': Original Process About to fork. ');
  intChildProcessID := fork;

  if intChildProcessID = 0 then
    begin
      //We are the child process
```

```

        writeln(getpid, ': Child Process Started.');
```

```

        setpgid(getpid, getppid);
    end else if intChildProcessID > 0 then
    begin
        writeln(getpid, ': Original Process fork.');
```

```

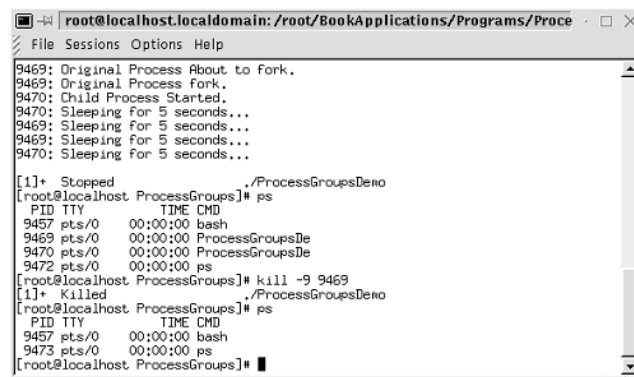
    end;

//We are the original process
for counter := 1 to 10 do
begin
    writeln(getpid, ': Sleeping for 5 seconds...');
```

```

    _sleep(5);
end;
end.
```

Figure 4.10 -
Using a Linux
command to
kill the
process group



```

root@localhost.localdomain: /root/BookApplications/Programs/Proce
File Sessions Options Help
9469: Original Process About to Fork.
9469: Original Process fork.
9470: Child Process Started.
9470: Sleeping for 5 seconds...
9469: Sleeping for 5 seconds...
9469: Sleeping for 5 seconds...
9470: Sleeping for 5 seconds...
[1]+  Stopped                  ./ProcessGroupsDemo
[root@localhost ProcessGroups]# ps
  PID TTY          TIME CMD
 9457 pts/0    00:00:00 bash
 9469 pts/0    00:00:00 ProcessGroupsDe
 9470 pts/0    00:00:00 ProcessGroupsDe
 9472 pts/0    00:00:00 ps
[root@localhost ProcessGroups]# kill -9 9469
[1]+  Killed                  ./ProcessGroupsDemo
[root@localhost ProcessGroups]# ps
  PID TTY          TIME CMD
 9457 pts/0    00:00:00 bash
 9473 pts/0    00:00:00 ps
[root@localhost ProcessGroups]#
```

Waiting on Other Processes

One of the common reasons for forking is to have a separate process execute operations that would often cause the original process to slow down in some way. You may also find that in order to solve a particular problem, you may need to solve several problems.

For example, you may have an application that obtains two stock prices, where one stock price is taken from one Web server and the other stock price is taken from another Web server. If you were to perform this operation sequentially, you would end up with a process that behaves like Figure 4.8.

Using sequential access to obtain the data introduces potential performance problems into the application. You may have a situation where the first of two servers takes 30 seconds to communicate and the second server takes 15 seconds to communicate. Assuming that the machine has enough network bandwidth when this operation is done sequentially as shown in Figure 4.8, the process will take around 45 seconds, depending on the time it takes to create the process.

If this operation is done using a forked process, the details can take less time, assuming that there is enough network connectivity to get the data. When the operation is done using a forked process, the application may fork once to set up a new child process to calculate the price of the first stock, and then the parent process may fork again to calculate the price of the second stock as shown in Figure 4.9. Using the forking technique, this process can take roughly 30 seconds, which would obviously be preferred.

In this example, now that we have forked the new processes, the parent process needs some way of determining when the child processes have finished their individual tasks.

Waiting for a process to finish after the process has forked is quite simple. Listing 4.6 demonstrates a simple process that forks and then waits for the child process to finish before it continues.

If you need to wait for a process to finish, you can use one of the many wait functions that are available under Linux. The premise of the wait functions are the same in that the function calls will not return until a particular process has been terminated, or in the case of some wait functions, until a child process has finished. Table 4.1 lists the typical wait API functions that you can use and what each function does.

Table 4.1 - The wait API functions

Function	Description
wait	This function will wait for any child process to finish.
waitpid	This function will wait for, depending on the settings, a particular process, a child process, or a member of the same process group to finish.
wait3	This function will wait for a child process to finish.
wait4	This function will wait for a child process to finish.

Listing 4.5 - Functions used when examining the result status of a wait function call

```
function WEXITSTATUS(Status: Integer): Integer;
function WTERMSIG(Status: Integer): Integer;
function WSTOPSIG(Status: Integer): Integer;
function WIFEXITED(Status: Integer): Boolean;
function WIFSIGNALED(Status: Integer): Boolean;
function WIFSTOPPED(Status: Integer): Boolean;
function WCOREDUMP(Status: Integer): Boolean;
```

Most of the wait functions will also return with a status describing what caused the process to terminate. In most cases, the process will terminate as a result of the process finishing its normal operations. But in some cases, the process may exit as a result of a signal being sent to the process and the process could not handle it, or a bad operation caused the process to have a core dump.

In the stock market download example, the forked processes that were created served a purpose to download a particular stock. If during the downloading of the stock quote the process was terminated by some method, use the return value from the waitpid function. This way the original process can see if it needs to fork again to attempt to download a stock price. Table 4.2 lists the functions that are used when examining the status result of the process and their purpose. Listing 4.6 demonstrates using the wait status functions to determine how a process ended.

Table 4.2 - Functions used to examine the reason a process terminated

Function	Description
WIFEXITED	The process ended through normal operation.
WEXITSTATUS	This function will return the child's exit value when the child calls the <code>__exit</code> function.
WTERMSIG	The process received the SIGTERM signal.

Function	Description
WSTOPSIG	The process received the SIGSTOP signal.
WIFSIGNALED	The process terminated because a signal was sent to the process that the process did not handle.
WIFSTOPPED	The process has been stopped.
WCOREDUMP	The process stopped suddenly and gave a core dump.

Listing 4.6 - Waiting for a process to finish

```

program ForkUsingWait;

uses
  Libc;

{$APPTYPE CONSOLE}

var
  CloneProcessID: integer;
  Status: integer;

begin
  writeln(getpid, ' (Parent): Starting the Cloning Process.');
```

CloneProcessID := fork;

```

  case CloneProcessID of
    0: begin
        //We are the cloned application
        Writeln(getpid, ' (Child): We are the Cloned Application.');
```

//Let's rest a little so that we can see that the parent process

//is waiting for us

```

        sleep(2);
        Writeln(getpid, ' (Child): Finished.');
```

end;

```

    -1: begin
        //There was an error in the forking process
        writeln(getpid, ' (Parent): An error occurred while forking.');
```

end;

```

  else
    begin
        //The fork was successful
        //Let's wait until the child has successfully done its thing.
        writeln(getpid, ' (Parent): Waiting for the process to finish');
```

waitpid(CloneProcessID, @Status, 0);

```

        //Write the reason the child process ended.
        write(getpid, ' (Parent): ');
        if WIFEXITED(Status) then
            writeln('The process ended normally.')
```

else if WEXITSTATUS(Status) <> 0 then

```

            writeln('The child was exited.')
```

else if WTERMSIG(Status) > 0 then

```

            writeln('The child retrieved the terminating signal')
```

else if WSTOPSIG(Status) > 0 then

```

            writeln('A signal caused the process to stop')
```

else if WIFSIGNALED(Status) then

```

        writeln('A signal caused the process to stop')
    else if WIFSTOPPED(Status) then
        writeln('The child is stopped.')
    else if WCOREDUMP(Status) then
        writeln('The process dumped its core.');
```

```

    writeln(getpid, ' (Parent): The child has finished.');
```

```

end;
end;
end.
```

Terminating a Process

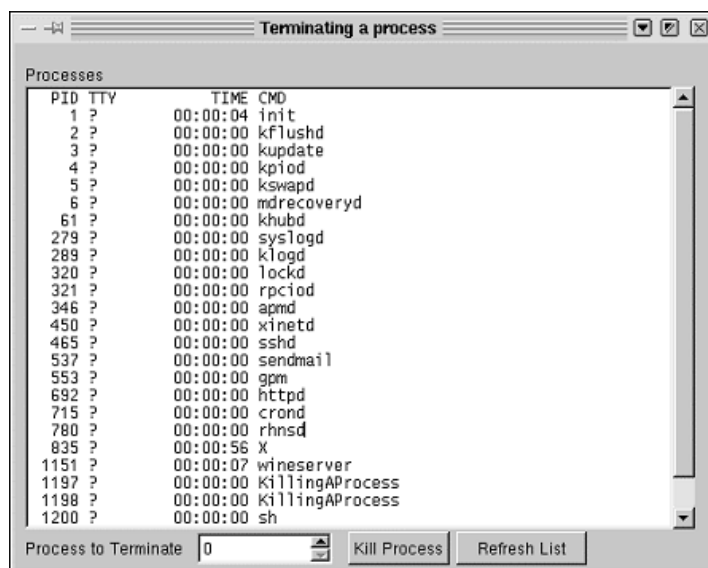
Terminating a process is a matter of using the kill function. The kill function is declared as follows:

```
function kill(ProcessID: __pid_t; SigNum: Integer): Integer; cdecl;
```

The kill function actually sends a signal to a process. Signals are covered in depth in Chapter 5, but signals essentially pass notifications of a particular event occurring that applies to the process. The signal is normally one of the signal constants. In the case of terminating a process, the signal that is passed is the SIGKILL constant, which will terminate a process.

Listing 4.7 demonstrates an application that displays a list of processes and allows you to terminate the process.

Figure 4.11 -
Example
application for
terminating a
process



Listing 4.7 - Terminating a simple process

```

unit unitMainForm;

interface
```

```

uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
  Libc, QStdCtrls, QComCtrls;

type
  TfrmMain = class(TForm)
    Label1: TLabel;
    mmProcessList: TMemo;
    Label2: TLabel;
    btnKillProcess: TButton;
    seProcessID: TSpinEdit;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure btnKillProcessClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    procedure UpdateProcessList;
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.xfm}

procedure TfrmMain.UpdateProcessList;
var
  strCommand: string;
begin
  strCommand := 'ps >' + extractFilePath(Application.ExeName) + 'currentProcesses.txt';
  Libc.system(PChar(strCommand));
  mmProcessList.Lines.LoadFromFile(extractFilePath(Application.ExeName) +
    'currentProcesses.txt');
end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  UpdateProcessList;
end;

procedure TfrmMain.btnKillProcessClick(Sender: TObject);
begin
  //kill the process using the kill function
  kill(seProcessID.Value, SIGKILL);

  //Now refresh the view of the current processes
  UpdateProcessList;
end;

procedure TfrmMain.Button1Click(Sender: TObject);
begin

```

```

    UpdateProcessList;
end;

end.

```

Executing Linux Commands

We discussed briefly in Chapter 3 the `system` command and how it can be used to execute a specific Linux command from within your application. The `system` command is a handy function that allows you to execute any valid Linux command in a single statement such as running a separate application or a single Linux command. As `system` is also the name of one of Kylix's main units, whenever you use the Linux API function `system` you must supersede it with the LibC unit name as shown in Listing 4.8.

Listing 4.8 - Using the LibC.system command

```

//Execute the command
strCommand := edCommand.Text;
Libc.system(PChar(strCommand));

```

One of the major advantages of doing this under Linux is that you can use the redirection of standard output to capture what a command does to a file so that you can analyze the output of the command. This can give your application the flexibility to obtain results from commands that may not be available with Kylix. An example of this technique can be seen in Listing 4.9.

Listing 4.9 - Executing a Linux command with LibC.system

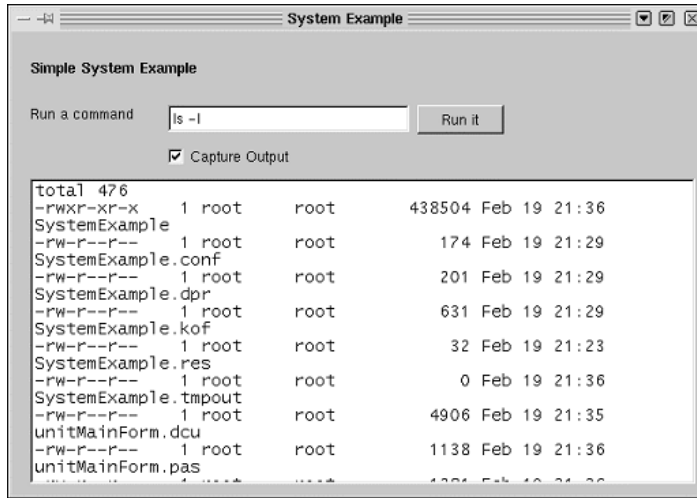
```

procedure TfrmMain.brnRunCommandClick(Sender: TObject);
var
    strTempFile: string;
    strCommand: string;
begin
    //Run the command
    if cbCaptureOutput.Checked then
    begin
        strTempFile := ChangeFileExt(Application.ExeName, '.tmpout');
        strCommand := edCommand.Text + ' > ' + strTempFile;
        Libc.system(PChar(strCommand));

        //Load the content from the generated output file
        mmOutput.Lines.LoadFromFile(strTempFile);
    end else begin
        //Execute the command
        strCommand := edCommand.Text;
        Libc.system(PChar(strCommand));
    end;
end;

```

Figure 4.12 -
Looking at the
output from a
system
command



The LibC.system function itself will create another process, execute the command, and wait for the command to finish before returning.

Using the exec Functions

A group of functions available under Linux is the exec functions. There are six of them, but are often only referred to as the exec function. The purpose of the exec function is to replace the current process by executing another process and keeping the same process ID.

There are six functions that can be used to execute this other process. They all perform the same basic task of replacing the process with a newly executed process, but differ slightly in the calling conventions of the different functions. Some of the functions require a full path to the file, whereas some of the functions will search on the user's path to find the file. Also, environment variables can be changed when executing the new application, which can be used to pass values by creating customized environment variables.

Listing 4.10 demonstrates a simple console application that executes another process. You should note that there are two writeln statements called prior to executing the new application and another line that is situated after the exec function call. As a call to exec will replace the current function, you can see from the output in Figure 4.13 that the second line will never be called.

Listing 4.10 - Using the execl function

```
program CallingAnExecFunction;

{$APPTYPE CONSOLE}

uses
  SysUtils, Libc;

(*
  This application demonstrates calling an exec
  function from a Linux application. The Exec functions
```

```

will replace the current running process with the
new process that you want to execute.

*)

var
  pFilenameToRun: array[0..250] of char;

begin
  if ParamCount = 1 then
    begin
      writeln('About to call execl to run the application.');
```

CallingAnExecFunction	CallingAnExecFunction.dpr	WriteDateAndTime.kof
CallingAnExecFunction.conf	WriteDateAndTime	WriteDateAndTime.dpr
CallingAnExecFunction.dpr	WriteDateAndTime.conf	
CallingAnExecFunction.kof	WriteDateAndTime.dpr	

```

      StrPCopy(pFilenameToRun, ParamStr(1));
      execl(pFilenameToRun, pFilenameToRun, nil);
      writeln('Run the program. This line should never be called.');
```

[root@localhost TheExecFunction]# ./CallingAnExecFunction ./WriteDateAndTime
--

```

    end else begin
      writeln('');
      writeln('One parameter was excepted.');
```

[root@localhost TheExecFunction]#

```

      writeln(' Usage '+ParamStr(0)+' <filename>');
      writeln('');
    end;
  end;
end.
```

Figure 4.13 -
Using the
execl function



Listing 4.11 - Using the exec functions

```

function execve(PathName: PChar; const argv: PPChar;
  const envp: PPChar): Integer; cdecl;
function execl(PathName: PChar; const argv: PPChar): Integer; cdecl;
function execl(_path: PChar; _arg: PChar): Integer; cdecl; varargs;
function execl(_path: PChar; _arg: PChar): Integer; cdecl; varargs;
function execvp(const FileName: PChar; const argv: PPChar): Integer; cdecl;
function execlp(_file: PChar; _arg: PChar): Integer; cdecl; varargs;
```

The execl function is perhaps one of the simplest of the exec functions to use. The function will take a parameter of the application to execute, a list of application arguments ending with nil for the newly created application, and the environment variables to use.

However, you may choose to use an exec function that is specific for your purpose. There is a standard in calling conventions when using the exec functions — the letters after

the functions define how the functions will be called. When the “l” is used after the “exec” prefix, it means that the arguments passed into the newly created process will be a list of values ending with a nil value. When the “v” character is used after the “exec” function, instead of a list of values that are being passed into the application, a PPChar vector that contains all the elements is required to be passed in. When the “e” character is used after the exec function, a variable that represents the environment variables that will be used by the new process is required to be passed in. In most cases, you may want to simply use the envp system variable that represents the current process’s path, or you could create your own interface.

The drawback when you are using a list of arguments in an exec function is that you need to know how many parameters you will have when calling the exec functions. If you have a variable amount of arguments or environment variables, the only option is to use the execve, execv, or execvp functions, as they allow the passing of a variable number of environment or argument variables. Creating a PPChar is not an easy task; the code in Listing 4.12 demonstrates a technique for converting the elements of a TStrings into a PPChar. When you use this technique for creating an argument vector, you should have the name of the application as the first parameter. This more closely resembles how the application works when executed normally. When using the StringListToPPChar function for environment, you should make sure that each line contains a name/value pair for each line to make up the environment values.

Finally, when the “p” character is used within the exec function, the file that is to be executed is required to be in the user’s path. The path is determined by the PATH environment variable on the system.

Listing 4.12 - Obtaining a vector from TStrings for use in the exec functions

```
program ExecUsingVectors;

{$APPTYPE CONSOLE}

uses
  Classes, SysUtils, Libc;

(*
   This application demonstrates using an exec
   function that requires a vector as a parameter
*)

var
  pFilenameToRun: array[0..250] of char;
  NewArguments: TStringList;
  counter: integer;

function StringListToPPChar(Value: TStrings): PPChar;
var
  i: integer;
  str: PChar;
  CompleteList: PPChar;
begin
  //This function converts a TStrings to a PPChar
```

```

//so that it can be used in functions such as execv

//Allocate the size of the environment list.
//We need to add one to allow for the null entry
CompleteList := malloc(sizeof(PChar) * (Value.Count+1));

//Store the original position for the result
Result := CompleteList;

//This function returns a custom environment for the new
//process, based on the name/value pairs within the memo.
for i := 0 to Value.Count - 1 do
begin
    str := malloc(Length(Value[i])+1);
    StrPCopy(str, Value[i]);

    //Now, let's add it into the pointer array
    CompleteList^ := str;
    inc(CompleteList);
end;

//Add the nil entry to the environment pointer list
//to confirm it is at the end.
CompleteList^ := nil;
end;

begin
    if ParamCount = 1 then
    begin
        writeln('About to call execv to demonstrate using vectors.');


NewArguments := TStringList.Create;



//Add the application name as the first parameter



NewArguments.Add(ParamStr(1));



for counter := 1 to 8 do



NewArguments.Add(inttostr(counter));



StrPCopy(pFilenameToRun, ParamStr(1));



execv(pFilenameToRun, StringListToPPChar(NewArguments));



writeln('Run the program. This line should never be called.');



end else begin



writeln('');



writeln('One parameter was excepted.');



writeln(' Usage '+ParamStr(0)+' <filename>');



writeln('');



end;



end.


```

Enumerating Users and Groups from within Code

If you are working with any kind of application that works with users or groups, such as an application that would impersonate another user, you may need to enumerate the users and groups that are currently running on your Linux system in the interface of your application.

Internally, the information about the file is stored in the `/etc/passwd` file as shown in Listing 4.13.

Listing 4.13 - An /etc/passwd file

```

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
news:x:9:13:news:/var/spool/news:
uucp:x:10:14:uucp:/var/spool/uucp:
operator:x:11:0:operator:/root:
games:x:12:100:games:/usr/games:
gopher:x:13:30:gopher:/usr/lib/gopher-data:
ftp:x:14:50:FTP User:/var/ftp:
nobody:x:99:99:Nobody:/:
apache:x:48:48:Apache:/var/www:/bin/false
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
gdm:x:42:42::/home/gdm:/bin/bash
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/bin/false
rpc:x:32:32:Portmapper RPC user:/bin/false
mailnull:x:47:47::/var/spool/mqueue:/dev/null
glenn:x:500:500:Glenn Stephens:/home/glenn:/bin/bash

```

Obtaining this list of users and groups from within the code is a matter of using the `getpwent` function. This function returns a pointer to a `TPasswordEntry` in the list. It continues returning the next entry from the `/etc/passwd` file until the last entry in the password file is reached and `nil` is returned as shown in Listing 4.14.

Listing 4.14 - Displaying a list of users in tree view

```

procedure TfrmListUsers.ReadInUsers;
var
  usr: PPasswordRecord;
  MyItem: TListItem;
begin
  //Read in the list of Users on the system
  usr := getpwent;
  while usr <> nil do
    begin
      //Display the Information About the User
      MyItem := lvUsers.Items.Add;
      MyItem.Caption := inttostr(usr^.pw_uid);
      MyItem.SubItems.Add(inttostr(usr^.pw_gid));
      MyItem.SubItems.Add(usr^.pw_name);
      MyItem.SubItems.Add(usr^.pw_dir);
      MyItem.SubItems.Add(usr^.pw_shell);

      usr := getpwent;
    end;
  end;
end;

```

Just as usernames and passwords are stored within the `/etc/passwd` file, the list of groups that are active on the system is stored in the `/etc/group` file. The function that is used to

return the entry to the group entry is `getgrnt`, which works in an identical manner to `getpwent` in that entries are returned one by one until all the entries have been obtained.

Listing 4.15 - Obtaining a list of groups from the `/etc/group` file

```

root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
tty:x:5:
disk:x:6:root
lp:x:7:daemon,lp
mem:x:8:
kmem:x:9:
wheel:x:10:root
mail:x:12:mail
news:x:13:news
uucp:x:14:uucp
man:x:15:
games:x:20:
gopher:x:30:
dip:x:40:
ftp:x:50:
nobody:x:99:
users:x:100:
apache:x:48:
utmp:x:22:
xfs:x:43:
floppy:x:19:
gdm:x:42:
pppusers:x:44:
popusers:x:45:
slipusers:x:46:
rpcuser:x:29:
rpc:x:32:
mailnull:x:47:
slocate:x:21:
glenn:x:500:

procedure TfrmListUsers.ReadInGroups;
var
    grp: PGroup;
    MyItem: TListItem;
begin
    //Read in the list of Groups in the system
    grp := getgrnt;
    while grp <> nil do
        begin
            //Display the Information about the Group
            MyItem := lvGroups.Items.Add;
            MyItem.Caption := inttostr(grp^.gr_gid);
            MyItem.SubItems.Add(grp^.gr_name);
            grp := getgrnt;
        end;
    end;
end;
```

Running Normal Apps with Permission of the App’s Owner

We have already looked at the fact that each process runs by a particular user. Consider the case of the `passwd` command under Linux. The `passwd` command modifies the password entries in the `/etc/passwd` file, which contains the passwords on the system. If you examine the file permissions as shown in Figure 4.14, you can see that the file can only be written to by the owner of the file, which in this case is the root user.

Figure 4.14 - Examining the permissions of the `/etc/passwd` file



So how does the `passwd` command update `/etc/passwd`? The trick is to use the `chmod` utility to specify that the application be given the permissions of the owner of the file. That way, when the application is run, it has permission to update the file.

Figure 4.15 - Looking at the permissions of the `passwd` application



Figure 4.15 demonstrates that the application allows any user to execute the application, but then it runs with the permission of the owner of the application, which in this case is the root user. The command to do this under Linux is:

```
chmod u+s <filename>
```

This will change the application to run under the privileges of the owner of the file. Only the owner of the file or a system administrator can perform this operation due to the likelihood of potential damage that can be done with the application. This is often known as setting the sticky bit for the user.

So the application will be run with two users. The first is the user that ran the application in the first place, known as the real user. The second is the user that the permission has changed to, known as the effective user.

From within your Kyx application, you already know how to obtain the user identifier using the `getuid` function. The `getuid` function actually returns the real user ID of the process. You can obtain the effective user ID of the application by using the `geteuid` function as shown in Listing 4.27.

This is an effective method of impersonating a user for a particular application. However, you may have a scenario where you have to impersonate many users, such as is the case with an FTP server. The FTP server obtains logins for each user and then only gives access to the file that a particular user has access to. In such a scenario, the server application may have to impersonate six to seven users and also the anonymous user. In such a case, what would most likely need to happen is that the application would fork a new process for each client connection and then change the effective user ID of the process by calling the `setuid` function. The `setuid` function will change the user the process is running under the permissions of if the process is currently running as the root user. Once the change has occurred, the process can never return to being owned by the root user.



Note: When you run an application with the permission of the file's owner, be extremely careful that the process does not perform any action that could create a security problem, such as calling a script name that was passed in as an application parameter. This could result in a malicious user running a script with the privileges of the superuser. One of the biggest problems that can occur is allowing your application to perform the system command to perform an action that results in giving someone access to some other file. Even a simple text editor when run with superuser privileges could give a user access to open the password file. These simple oversights are often the way many hackers obtain access to areas where they should not be.

Creating Daemons for Linux

A daemon is an invisible process that starts when Linux runs and will normally provide a service of some sort such as FTP, the Apache Web server, or remote procedure calls to users.

Linux daemons have some special attributes that do not apply to normal processes. Daemons are active the entire time that the Linux machine is operational and will run with the permissions of the superuser system.

A daemon is started by the `init` process when Linux is started and orphans itself by forking the process and killing the original parent process. Another interesting point about

daemons running on Linux is that unlike normal processes they should not have access to standard input, standard output, and standard error. Daemons are supposed to be silent; they should not affect the normal operation of applications so you should close the standard input, output, and error file descriptors as soon as possible.

As all input and output streams are closed by a daemon, there is normally no place to write error messages as the standard error is already closed. Under these circumstances, errors handled within daemon application are either written to a specific log file or to the system log. In the case of a specific log, you would normally have a subdirectory for your application in the /etc directory and write the log entries into a specific file there. There is also the option to write the values into the system log using syslog.

The last setup point for a daemon is to change the directory where the daemon lives. Normally the directory that you change to is the “/” directory. Changing to the / directory removes the chance of your daemon starting in a directory which may be a device directory and would cause unnecessary hard disk access. This is not the only directory that your daemon can change to. If you are writing log entries to a specific directory, it is a good idea to change the directory to the directory where the log entries are being written to.

Once all of the housekeeping of setting up the daemon is done, you can get your daemon to do all the fun and exciting things that you want it to.

One of the main things to keep in mind when writing daemons is to not make them too process intensive as in many cases they have to work at the same time as many applications and services and your daemon should not detract from the performance of any other application or daemon. As a result, your daemon should do the processing that it wants to perform and then wait until it is needed again before processing any more information. It can perform the waiting function by using the sleep function.

To pull this all together the daemon that we come up with should look like Listing 4.16.

Listing 4.16 - A template that can be used for most daemons

```
program SimpleDaemon;

uses
  Libc;

{$APPTYPE CONSOLE}

var
  pid, sid: integer;
  Done: boolean;

procedure LogErrorMessage(sMessage: string);
const
  LOGFILE = '/';
begin
  //Open the Log for the daemon and include the Process ID
  //in the log entry
  openlog(LOGFILE, LOG_PID, LOG_DAEMON);
  //Write the message out
  syslog(LOG_ERR, PChar(sMessage));
  //Close the log
```

```

    closelog;
end;

begin
    //Fork the process, so we can have an orphan daemon
    pid := fork;

    //See if there was a problem forking
    if pid = -1 then
    begin
        //There was an error forking
        LogErrorMessage('fork creation');
        Exit;
    end;

    if pid > 0 then
    begin
        //We are the original process, so we need
        //to terminate the parent process
        Exit;
    end;

    //If the application gets to here it means
    //that this process will be the one that lives

    //Start a new Session
    sid := setsid;
    if sid < 0 then
    begin
        LogErrorMessage('Set Session ID Failed');
        Exit;
    end;

    //Change the directory to the / dir.
    //This ensures there is no lag time which
    //may be caused by trying to access devices
    _chdir('/');

    //Set the umask for the creation of any new files
    umask(0);

    //Close all the streams
    _close(STDIN_FILENO);
    _close(STDOUT_FILENO);
    _close(STDERR_FILENO);

    //Now here is where all the work for the
    //daemon occurs.
    Done := false;
    while not Done do
    begin
        //Do some action. Sleep until the next
        //operation needs performing
        //LogErrorMessage('The daemon is now running');
        //break;

        sleep(1000);
    end;
end;

```

```
end;
end.
```



Note: For developers coming from a Windows background, the daemon is similar to Windows NT services in the way that it provide services to the system and is always active on the machine.

API Reference

`__secure_getenv` *LibC.pas*

Syntax

```
function __secure_getenv(
  Name: PChar;
):PChar;
```

Description

The `__secure_getenv` function retrieves an environment value in the same way `getenv` is used. `__secure_getenv`, however, will return a nil value when the user is a superuser, such as root.

When an application has changed the effective user ID of an application to run with the same permission as the root user, calling `getenv` may retrieve an environment variable that may have a sensitive value. Using the `__secure_getenv` ensures that this will not occur.

Parameters

Name: Name refers to the name of the environment value in the name/value pair.

Return Value

The function returns the value of the environment name/value pair unless the application is running with the effective user ID or effective group ID set to the root user.

See Also

`getenv`, `setenv`, `putenv`, `clearenv`

Example

Listing 4.17 - Using the `__secure_getenv` function

```
var
  MailDir: PChar;

begin
  //Find out the location of the Mail directory.
  MailDir := __secure_getenv('MAIL');
  if MailDir <> nil then
    showmessage('Mail for the user can be found in '+StrPas(MailDir))
  else
```

```
    showmessage('You do not have access to this method.');
```

```
end;
```

clearenv ***LibC.pas***

Syntax

```
function clearenv: Integer;
```

Description

The `clearenv` function clears the environment name/value pairs list. This function only persists to the current process and any child processes that are created from the application using the `fork` or `exec` functions.

Return Value

The function returns 0 if the environment variables are cleared or -1 if the environment variables were not cleared.

See Also

`getenv`, `setenv`, `putenv`, `__secure_getenv`

Example

Listing 4.18 - Using the `clearenv` function

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    //Clear the environment variables
    if clearenv = 0 then
        showmessage('The environment is cleared.')
    else
        showmessage('The environment could not be cleared.');
```

```
end;
```

daemon ***LibC.pas***

Syntax

```
function daemon(
    NoChDir: Integer;
    NoClose: Integer
):Integer;
```

Description

The `daemon` function is called to set up a process to behave like a daemon. It will make the process not have a controlling terminal and also gives the option of closing standard input, output, and error as well as the option to change the directory.

Parameters

NoChDir: When this value is non-zero, this will change the process's current directory to the / directory. In the case of daemons, this is preferred so the process will not unintentionally access devices and slow the processing down.

NoClose: When this value is non-zero, the process will close standard input, standard output, and standard error.

Return Value

If the function was unsuccessful, the function returns -1, and the reason for the error can be accessed by calling the `errno` function.

See Also

`fork`

Example**Listing 4.19 - Using the daemon function**

```
//Make the process run in the background and
//change the directory to '/' and close
//standard input, output and error
daemon(1, 1);
```

endgrent LibC.pas**Syntax**

```
procedure endgrent;
```

Description

The `endgrent` function signifies that the application has finished reading group entries from the `/etc/group` file and that the file should be closed.

See Also

`getgrent`, `setgrent`, `getgrgid`, `getgrnam`

Example

See Listing 4.29 - Using the `getgrent` function.

endpwent LibC.pas**Syntax**

```
procedure endpwent;
```

Description

The `endpwent` function signifies that the application has finished working with the `/etc/passwd` file and that the file should be closed.

See Also

getpwent, setpwent

Example

See Listing 4.37 - Using the getpwent function.

execl LibC.pas**Syntax**

```
function execl(
  __path: PChar;
  __arg: PChar;
):Integer; cdecl; varargs;
```

Description

The execl function replaces the current process with the application passed to the __path parameter. After the __path parameter, the function requires that the arguments to the application be included, ending with a nil parameter. The first argument should be the application name to emulate how arguments are passed to the application from the Linux shell.

Parameters

__path: This is the full pathname to the application that will replace the current process.
 __arg: This is the first parameter to pass to the new application. Other arguments for the new application should be passed as varargs arguments with a nil pointer denoting that there are no more parameters to the application.

Return Value

This function will only return a value if the function fails, in which case the return value will be -1. If the function is successful, the new original process will be replaced by the new application.

See Also

execlp, execl, execv, execvp, execve

Example

See Listing 4.10 - Using the execl function.

execle LibC.pas**Syntax**

```
function execle(
  __path: PChar;
  __arg: PChar;
):Integer;
```

Description

The `execle` function replaces the current process with the application passed to the `__path` parameter and defines the application environment name/value pairs to the new application. After the `__path` parameter, the function requires the arguments to the application to be included, ending with a `nil` parameter and then a pointer to an array of `PChar`, which holds the environment values.

The first argument should be the application name to emulate how arguments are passed to the application from the Linux shell.

Parameters

`__path`: This is the full pathname to the application that will replace the current process.

`__arg`: This is the first parameter to pass to the new application, which should be the same value that is passed into the `__path` parameter. Other arguments for the new application should be passed as `varargs` arguments with a `nil` pointer denoting that there are no more parameters to the application.

`varargs`: The `varargs` parameters for this application should be all the arguments after the program name followed by a `nil` value to denote the end of the arguments.

Return Value

This function will only return a value if the function fails, in which case the return value will be `-1`. If the function is successful, the original process will be replaced by the new application.

See Also

`execl`, `execlp`, `execv`, `execvp`, `execve`

*Example***Listing 4.20 - Using the `execle` function**

```
program UsingTheExecleFunction;

{$APPTYPE CONSOLE}

uses
  Classes, SysUtils, Libc;

var
  pFilenameToRun: array[0..250] of char;

begin
  if ParamCount = 1 then
    begin
      writeln('About to call execle...');
      StrPCopy(pFilenameToRun, ParamStr(1));
      execle(pFilenameToRun, pFilenameToRun, envp);
    end else begin
      writeln('');
      writeln('One parameter was excepted.');
```

```

        writeln(' Usage '+ParamStr(0)+' <filename>');
        writeln('');
    end;
end.

```

execlp LibC.pas

Syntax

```

function execlp(
    __path: PChar;
    __arg: PChar;
):Integer;

```

Description

The `execlp` function replaces the current process with the application passed to the `__path` parameter. After the `__path` parameter, the function requires the arguments to the application to be included, ending with a `nil` parameter. The first argument should be the application name to emulate how arguments are passed to the application from the Linux shell. The only difference between this function and `execl` is that the path argument will look for the application in the path if the application cannot be found.

Parameters

`__path`: This is either the full pathname to the application that will replace the current process or it is the name of an application that is located on the path.

`__arg`: This is the first parameter to pass to the new application. Other arguments for the new application should be passed as `varargs` arguments with a `nil` pointer denoting that there are no more parameters to the application.

Return Value

This function will only return a value if the function fails, in which case the return value will be `-1`. If the function is successful, then the original process will be replaced by the new application.

See Also

`execl`, `execle`, `execv`, `execvp`, `execve`

Example

Listing 4.21 - Using the `execlp` function

```

program UsingTheExeclpFunction;

{$APPTYPE CONSOLE}

uses
    Classes, SysUtils, Libc;

```

```

var
  pFilenameToRun: array[0..250] of char;

begin
  if ParamCount = 1 then
    begin
      //This demonstrates using the path with the exec functions
      //This application should be called with something like
      //  ./UsingtheExeclpFunction ps
      writeln('About to call execvp...');
      StrPCopy(pFilenameToRun, ParamStr(1));
      execlp(pFilenameToRun, pFilenameToRun, nil, envp);
    end else begin
      writeln('');
      writeln('One parameter was excepted.');
```

```

      writeln(' Usage '+ParamStr(0)+' <filename>');
      writeln('');
    end;
  end.

```

execv LibC.pas

Syntax

```

function execv(
  PathName: PChar;
  const argv: PPChar;
):Integer;
```

Description

The `execv` function replaces the current process with the application passed to the `__path` parameter. After the `__path` parameter, the function requires that the arguments to the application be included as an array of `PChar`, which is also termed a vector of arguments.

Parameters

PathName: This is either the full pathname to the application that will replace the current process, or the name of an application on the current path that will be executed.

argv: This is the vector of arguments for the application that will replace the current process. The first entry in this list should be the same value as `PathName` to emulate how arguments are passed to the application by the Linux shell.

Return Value

This function will only return a value if the function fails, in which case the return value will be `-1`. If the function is successful, the original process will be replaced by the new application.

See Also

`execl`, `execlp`, `execle`, `execvp`, `execve`

*Example***Listing 4.22 - Using the `execv` function**

```

program ExecUsingVectors;

{$APPTYPE CONSOLE}

uses
  Classes, SysUtils, Libc;

(*
   This application demonstrates using an exec
   function that requires a vector as a parameter
*)

var
  pFilenameToRun: array[0..250] of char;
  NewArguments: TStringList;
  counter: integer;

function StringListToPPChar(Value: TStrings): PPChar;
var
  i: integer;
  str: PChar;
  CompleteList: PPChar;
begin
  //This function converts a TStrings to a PPChar
  //so that it can be used in functions such as execv

  //Allocate the size of the environment list.
  //We need to add one to allow for the null entry
  CompleteList := malloc(sizeof(PChar) * (Value.Count+1));

  //Store the original position for the result
  Result := CompleteList;

  //This function returns a custom environment for the new
  //process, based on the name/value pairs within the memo.
  for i := 0 to Value.Count - 1 do
  begin
    str := malloc(Length(Value[i])+1);
    StrPCopy(str, Value[i]);

    //Now, let's add it into the pointer array
    CompleteList^ := str;
    inc(CompleteList);
  end;

  //Add the nil entry to the environment pointer list
  //to confirm it is at the end.
  CompleteList^ := nil;
end;

begin
  if ParamCount = 1 then
    begin

```

```

writeln('About to call execv to demonstrate using vectors.');
```

```

NewArguments := TStringList.Create;
NewArguments.Add(ParamStr(1));
for counter := 1 to 8 do
    NewArguments.Add(inttostr(counter));
StrPCopy(pFilenameToRun, ParamStr(1));

execv(pFilenameToRun, StringListToPPChar(NewArguments));

writeln('Run the program. This line should never be called.');
```

```

end else begin
writeln('');
writeln('One parameter was excepted.');
```

```

writeln(' Usage '+ParamStr(0)+' <filename>');
writeln('');
end;
end.
```

execve *LibC.pas*

Syntax

```

function execve(
  PathName: PChar;
  const argv: PPChar;
  const envp: PPChar;
):Integer;
```

Description

The `execve` function replaces the current process with the application passed to the `__path` parameter and passes arguments to the application as well as establishing the environment variables for the new application. After the `__path` parameter, the function requires that the arguments to the application be included as an array of `PChar`, which is also termed a vector of arguments. Another vector of name/value pairs should also be included to denote the environment variables for that new application.

Parameters

`PathName`: This is the full pathname to the application that will replace the current process.

`argv`: This is the vector of arguments for the application that will replace the current process. The first entry in this list should be the same value as `PathName` to emulate how arguments are passed to the application by the Linux shell.

`envp`: This is another vector of arguments that represents the environment variables. Each entry in this vector should be a string in the format "name=value".

Return Value

This function will only return a value if the function fails, in which case the return value will be `-1`. If the function is successful, the original process will be replaced by the new application.

See Also

execl, execlp, execl, execvp, execv

Example

Listing 4.23 - Using the execve function

```

program ExecUsingVectorsAndEnvironment;

{$APPTYPE CONSOLE}

uses
  Classes,
  SysUtils,
  Libc;

(*
   This application demonstrates using an exec
   function that requires a vector as a parameter
   and also sends the environment variables
*)

var
  pFilenameToRun: array[0..250] of char;
  NewArguments: TStringList;
  NewEnvironment: TStringList;
  counter: integer;

function StringListToPPChar(Value: TStrings): PPChar;
var
  i: integer;
  str: PChar;
  CompleteList: PPChar;
begin
  //This function converts a TStrings to a PPChar
  //so that it can be used in functions such as execv

  //Allocate the size of the environment list.
  //We need to add one to allow for the null entry
  CompleteList := malloc(sizeof(PChar) * (Value.Count+1));

  //Store the original position for the result
  Result := CompleteList;

  //This function returns a custom environment for the new
  //process, based on the name/value pairs within the memo.
  for i := 0 to Value.Count - 1 do
  begin
    str := malloc(Length(Value[i])+1);
    StrPCopy(str, Value[i]);

    //Now, let's add it into the pointer array
    CompleteList^ := str;
    inc(CompleteList);
  end;
end;

```



```

//Add the nil entry to the environment pointer list
//to confirm it is at the end.
CompleteList^ := nil;
end;

begin
  if ParamCount = 1 then
    begin
      writeln('About to call execve to demonstrate using vectors. ');
      NewArguments := TStringList.Create;
      NewArguments.Add(ParamStr(1));
      for counter := 1 to 8 do
        NewArguments.Add(inttostr(counter));
      StrPCopy(pFilenameToRun, ParamStr(1));

      NewEnvironment := TStringList.Create;
      NewEnvironment.Add('book=Tomes of Kylix');
      NewEnvironment.Add('author=Glenn Stephens');
      NewEnvironment.Add('publishedby=Wordware');

      execve(pFilenameToRun, StringListToPPChar(NewArguments),
        StringListToPPChar(NewEnvironment));

      writeln('Run the program. This line should never be called. ');
    end else begin
      writeln(' ');
      writeln('One parameter was excepted. ');
      writeln(' Usage '+ParamStr(0)+' <filename> ');
      writeln(' ');
    end;
  end.

```

execvp LibC.pas

Syntax

```

function execvp(
  const FileName: PChar;
  const argv: PPChar;
):Integer;

```

Description

The `execvp` function replaces the current process with an application denoted in the `FileName` parameter, where the application that is replacing the file is run with the parameters passed to `argv`. If `FileName` is a relative filename, it should be an application that currently reside on the user's search path.

Parameters

FileName: This is either the full pathname to the application that will replace the current process or the name of an application that is located on the path.

argv: This is the vector of arguments for the application that will replace the current process. The first entry in this list should be the same value as `FileName` to emulate how arguments are passed to the application by the Linux shell.

Return Value

This function will only return a value if the function fails, in which case the return value will be -1 . If the function is successful, the original process will be replaced by the new application.

See Also

`execl`, `execlep`, `execle`, `execv`, `execve`

Example

Listing 4.24 - Using the `execvp` function

```
begin
  if ParamCount = 1 then
    begin
      writeln('About to call execvp to demonstrate using vectors.');
```

```
      NewArguments := TStringList.Create;
      NewArguments.Add(ParamStr(1));
      for counter := 1 to 8 do
        NewArguments.Add(inttostr(counter));
      StrPCopy(pFilenameToRun, ParamStr(1));

      execvp(pFilenameToRun, StringListToPPChar(NewArguments));

      writeln('Run the program. This line should never be called.');
```

```
    end else begin
      writeln('');
      writeln('One parameter was excepted.');
```

```
      writeln(' Usage '+ParamStr(0)+' <filename>');
```

```
      writeln('');
    end;
  end.
```

fexecve *LibC.pas*

Syntax

```
function fexecve(
  FileDes: Integer;
  const argv: PPChar;
  const envp: PPChar;
):Integer;
```

Description

This function replaces the current process with a new process denoted by the file that is pointed to by the `FileDes` file descriptor. This application, which replaces the process, also executes as if the application started with the arguments in `argv` and the environment values found in `envp`.

Parameters

FileDes: This represents a current open file descriptor that was created using the open function. The actual file that is represented by this file descriptor will be used as the application that replaces the current process.

argv: This is the vector of arguments for the application that will replace the current process.

envp: This is another vector of arguments that represents the environment variables. Each entry in this vector should be a string in the format "name=value".

Return Value

This function will only return a value if the function fails, in which case the return value will be -1 . If the function is successful, the original process will be replaced by the new application.

See Also

execl, execlp, execle, execvp, execv

Example

Listing 4.25 - Using the fexecve function

```
begin
  if ParamCount = 1 then
    begin
      writeln('About to call fexecve to demonstrate using vectors.');
```

```
      NewArguments := TStringList.Create;
      NewArguments.Add(ParamStr(1));
      for counter := 1 to 8 do
        NewArguments.Add('Argument'+inttostr(counter));
      StrPCopy(pFilenameToRun, ParamStr(1));

      AppFileDescriptor := open('/usr/glenn/MydemoApplication', 666);
      if AppFileDescriptor <> -1 then
        fexecve(AppFileDescriptor, StringListToPPChar(NewArguments), envp);
    end else begin
      writeln('');
      writeln('One parameter was excepted.');
```

```
      writeln(' Usage '+ParamStr(0)+' <filename>');
      writeln('');
    end;
  end.
```

fork LibC.pas

Syntax

```
function fork: __pid_t;
```

Description

The fork function creates a copy of the current process, including the location of which instruction to execute next. The fork function itself will return twice, once in the original

process and once in the copy of the process. The fork function is often used to pass off functionality to a separate process so that the newly created child process can handle the functionality and the original process can handle incoming requests before passing them off. The new process will have a new process ID and its parent process returned by the getpid function, which will return the process ID of the original process.

Return Value

If the function is successful, then the original process will return the new process ID of the child process and the child process will return 0. If the function fails, the function will return -1 in the parent process. The function will normally only fail if there is insufficient memory to fork the process.

See Also

getpid, getppid

Example

See Listing 4.3 - A simple forking application.

getegid LibC.pas

Syntax

function getegid: __gid_t

Description

This function returns the effective group ID of the current process. The effective group is normally set as a result of the process setting the sticky group bit on the permission mask for the application, so that when the process runs, it will run with the group permission of the owner of the application file.

Return Value

The function returns the effective group ID of the process. This function will never return an error.

See Also

getuid, geteuid, seteuid, setegid

Example

See Listing 4.28 - Using the getgid function.

getenv LibC.pas

Syntax

```
function getenv(
  Name: PChar
):PChar;
```

Description

The `getenv` function returns the value of an environment name/value pair. A typical environment variable is of the form `HOME=/root`. A list of environment variables can typically be viewed by using the `env` command from the Linux terminal.

Parameters

Name: Name refers to the name of the environment value in the name/value pair.

Return Value

The function returns the value of the environment name/value pair.

See Also

`setenv`, `putenv`, `clearenv`, `__secure_getenv`

*Example***Listing 4.26 - Using the `getenv` function**

```
var
    MailDir: string;

begin
    //Find out the location of the Mail directory.
    MailDir := StrPas(getenv('MAIL'));
    showmessage('Mail for the user can be found in '+MailDir);
end;
```

geteuid* LibC.pasSyntax*

```
function geteuid: __uid_t;
```

Description

The `geteuid` function returns the effective user ID of the current running process. The effective user ID is set as a result of the user sticky bit set on the application being run.

Return Value

This function returns the effective user ID of the application. The result is often passed into `getpwuid` to obtain more information about the user.

See Also

`getuid`, `getgid`, `getegid`, `getpwuid`

*Example***Listing 4.27 - Displaying user information about a process**

```
procedure TfrmMain.DisplayUserInformation;
var
    pwInfo: PPasswordRecord;
```

```

begin
    //Display the User ID
    pwInfo := getpwuid(getuid);
    mmUserInformation.Lines.Add('The Real User ID of the application is '+
        pwInfo^.pw_name);
    //Display the Effective User ID
    pwInfo := getpwuid(geteuid);
    mmUserInformation.Lines.Add('The Effective User ID of the application is '+
        pwInfo^.pw_name);
end;

```

getgid *LibC.pas*

Syntax

```
function getgid: __gid_t;
```

Description

The `getgid` function returns the real group ID of the process that is currently running. The real group ID is normally set by the owner of the file so that members of the groups have their own set of permissions to the file. For more information about file permissions, see Chapter 3.

Return Value

The function represents the group ID of the process. The result is normally passed into `getgrgid` to obtain a person-friendly name of the group.

See Also

`getegid`, `getuid`, `geteuid`, `getgrgid`

Example

Listing 4.28 - Using the `getgid` function

```

procedure TfrmMain.DisplayGroupInformation;
var
    grpInfo: PGroup;
begin
    //Display the Real Group
    grpInfo := getgrgid(getgid);
    mmUserInformation.Lines.Add('The Real Group ID of the application is '+
        grpInfo^.gr_name);
    //Display the Effective Group
    grpInfo := getgrgid(getegid);
    mmUserInformation.Lines.Add('The Effective Group ID of the application is '+
        grpInfo^.gr_name);
end;

```

getgrent *LibC.pas*

Syntax

```
function getgrent: PGroup;
```

Description

The `getgrent` function will return a pointer to a `TGroup` record that represents one entry from the `/etc/group` file.

```
type
  group = record
    gr_name: PChar;           // Group name.
    gr_passwd: PChar;         // Password.
    gr_gid: __gid_t;          // Group ID.
    gr_mem: PPChar;           // Member list
  end;
  TGroup = group;
  PGroup = ^TGroup;
```

When this function is called for the first time, the first entry from the `/etc/group` file will be read, and every successive call will read the next entry from the `/etc/group` file. Once all the entries have been read, only a call to the `setgrent` function will reset the reading of group entries to the beginning of the `/etc/group` file.

Return Value

If the function is successful, the function will return a pointer to a `TGroup` structure that represents a group on the system. If there are no more entries to read in the file or the function results in an error, the function will return `nil`.

See Also

`setgrent`, `endgrent`, `getgrid`, `getgrname`

*Example***Listing 4.29 - Using the `getgrent` function**

```
procedure TfrmUsers.ListGroups(AStrings: TStrings);
var
  grp: PGroup;
begin
  AStrings.Clear;
  //Reset the /etc/group file read position
  setgrent;
  try
    //Read all the entries
    grp := getgrent;
    while grp <> nil do
      begin
        AStrings.Add(grp^.gr_name + '(' +
          IntToStr(grp^.gr_gid) + ')');
        grp := getgrent;
      end;
    finally
      endgrent;
    end;
  end;
end;
```

getgrgid LibC.pas**Syntax**

```
function getgrgid(
  gid: __gid_t
):PGroup;
```

Description

The `getgrgid` function will return the details about a particular group from the `/etc/group` file when only the ID of the group is known. The information about the user will then be returned as a pointer to a `TGroup` that represents the group.

```
type
  group = record
    gr_name: PChar;           // Group name.
    gr_passwd: PChar;         // Password.
    gr_gid: __gid_t;          // Group ID.
    gr_mem: PPChar;           // Member list
  end;
  TGroup = group;
  PGroup = ^TGroup;
```

Parameters

`gid`: This parameter is the integer identifier of the group that is stored in the `/etc/group` file.

Return Value

If the group identifier passed to the `gid` parameter is valid, then the function will return a pointer to a `TGroup` record which signifies the details of the group. If the `gid` parameter does not represent a valid user, or an error occurred while using the function, the function will return `nil`.

See Also

`setgrent`, `endgrent`, `getgrent`, `getgrnam`

Example**Listing 4.30 - Using the `getgrgid` function**

```
procedure TfrmUsers.FindGroupByID;
var
  grp: PGroup;
begin
  mmSearchResults.Lines.Clear;
  grp := getgrgid(StrToInt(edSearchValue.Text));
  if grp = nil then
    mmSearchResults.Lines.Add('No Group by that ID found.')
  else begin
    mmSearchResults.Lines.Add('Group ID: '+inttostr(grp^.gr_gid));
    mmSearchResults.Lines.Add('Group Name: '+grp^.gr_name);
    mmSearchResults.Lines.Add('Group Password: '+grp^.gr_passwd);
  end;
end;
```


getgrnam LibC.pas**Syntax**

```
function getgrnam(
  Name: PChar
):PGroup;
```

Description

The `getgrnam` function will return the details about a group from the `/etc/group` file when only the name of the group is known. The function will then return the details in the form of a pointer to a `TGroup` record that represents the stored information in the `/etc/group` file for the group associated with the group.

```
type
  group = record
    gr_name: PChar;           // Group name.
    gr_passwd: PChar;        // Password.
    gr_gid: __gid_t;         // Group ID.
    gr_mem: PPChar;          // Member list
  end;
  TGroup = group;
  PGroup = ^TGroup;
```

Parameters

Name: This parameter is the name of the group that you are searching for.

Return Value

If the **Name** parameter signifies a valid group, the function will return a pointer to a `TGroup` record, which contains the details about the group. If the group could not be found or an error occurs during the function call, then the function will return `nil`.

See Also

`setgrent`, `endgrent`, `getgrent`, `getgrgid`

Example**Listing 4.31 - Using the `getgrnam` function**

```
procedure TfrmUsers.FindGroupName;
var
  grp: PGroup;
  strGroupName: string;
begin
  mmSearchResults.Lines.Clear;
  strGroupName := edSearchValue.Text;
  grp := getgrnam(PChar(strGroupName));
  if grp = nil then
    mmSearchResults.Lines.Add('No Group by that Name found.')
  else begin
    mmSearchResults.Lines.Add('Group ID: '+inttostr(grp^.gr_gid));
    mmSearchResults.Lines.Add('Group Name: '+grp^.gr_name);
    mmSearchResults.Lines.Add('Group Password: '+grp^.gr_passwd);
```

```
end;
end;
```

getgroups *LibC.pas*

Syntax

```
function getgroups(
  Size: Integer;
  List: PGid;
):Integer;
```

Description

This function will return a list of the groups of which the current process is a member.

Parameters

Size: This is the size of the buffer of group IDs that will be written to the List parameter buffer. If this value is 0, then no group IDs will be written into the List parameter, but rather the number of groups will be returned by the function.

List: This is a pointer to a buffer that will hold the list of groups that this process belongs to. This buffer should hold at least the amount of entries that are in the Size parameter.

Return Value

When successful, this function returns the number of entries written to the List parameter. If the Size parameter is 0, the function will simply return the amount of groups of which the current process is a member.

See Also

getgrent, getpwent

Example

Listing 4.32 - Using the getgroups function

```
procedure TfrmLinuxGroups.Button1Click(Sender: TObject);
var
  counter: integer;
  GroupID: integer;
  AllGroups: array[0..500] of __pid_t;
  GroupCount: integer;
begin
  //this will display up to 500 groups, although it is unlikely that
  //the 500 limit will be reached.
  GroupCount := getgroups(500, @AllGroups);

  clbGroups.Items.Clear;
  for counter := 0 to GroupCount - 1 do
    clbGroups.Items.Add(GroupIDToGroupName(AllGroups[counter]));
  end;
```

getpgid LibC.pas**Syntax**

```
function getpgid(
  ProcessID: __pid_t;
): __pid_t;
```

Description

This function returns the process group ID of the process identified by the process ID passed into the parameter ProcessID.

Parameters

ProcessID: This is a process ID of which to return the process group ID. If this value is 0, this function returns the process group ID of the current process.

Return Value

When successful, this function returns the process group ID of the process, which should be a non-negative integer. If the function failed, it will return -1, and is normally a result of the ProcessID parameter not being a valid process ID.

See Also

getpid, setpgid

Example**Listing 4.33 - Using the getpgid function**

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  mmProcessGroupInfo.Lines.Add('The current Process is '+
    inttostr(getpid));
  mmProcessGroupInfo.Lines.Add('The current Process Group is '+
    inttostr(getpgid(getpid)));
  if getpgid(getpid) = getpid then
    mmProcessGroupInfo.Lines.Add('This process is the process group leader.');
```

end;

getpgrp LibC.pas**Syntax**

```
function getpgrp: __pid_t;
```

Description

This function returns the process group ID of the current process.

Return Value

This function always returns a positive value that represents the process group ID of the current process.

See Also

getpid, getppid, setpgid

*Example***Listing 4.34 - Using the getpgrp function**

```

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  mmProcessGroupInfo.Lines.Add('The current Process is '+
    inttostr(getpid));
  mmProcessGroupInfo.Lines.Add('The current Process Group is '+
    inttostr(getpgrp));
  if getpgid(getpid) = getpid then
    mmProcessGroupInfo.Lines.Add('This process is the process group leader.');
```

getpid LibC.pas*Syntax*

```
function getpid: __pid_t;
```

Description

This function returns the unique identifier for this process known as the process ID. Each process ID is created by the kernel and is guaranteed to be unique.

Return Value

This function will always return a positive integer value that represents the process identifier. This function will never result in an error.

See Also

getppid

*Example***Listing 4.35 - Using the getpid function**

```

procedure TfrmProcessInfo.FormCreate(Sender: TObject);
begin
  mmProcessInfo.Lines.Add('Process ID is '+inttostr(getpid));
end;
```

getppid LibC.pas*Syntax*

```
function getppid: __pid_t;
```

Description

This function returns the process ID of the current process. Each process has a parent process, except the kernel which has a process ID of 0.

Return Value

This function will always return an integer value that represents the process identifier. This function will never result in an error.

See Also

getpid, fork

Example

Listing 4.36 - Using the getppid function

```
procedure TfrmProcessInfo.FormCreate(Sender: TObject);
begin
    mmProcessInfo.Lines.Add('Process ID is '+inttostr(getpid));
    mmProcessInfo.Lines.Add('Parent Process ID is '+inttostr(getppid));
end;
```

getpwent LibC.pas

Syntax

```
function getpwent:PPasswordRecord;
```

Description

The getpwent function will return a pointer to a TPasswordRecord record that represents one entry from the /etc/passwd file.

```
type
    passwd = record
        pw_name: PChar;           // Username
        pw_passwd: PChar;        // Password
        pw_uid: __uid_t;         // User ID
        pw_gid: __gid_t;         // Group ID
        pw_gecos: PChar;         // Real name
        pw_dir: PChar;           // Home directory
        pw_shell: PChar;         // Shell program.
    end;
    TPasswordRecord = passwd;
    PPasswordRecord = ^TPasswordRecord;
```

When this function is called for the first time, the first entry from the /etc/passwd file will be read, and every successive call will read the next entry. Once all the entries have been read, only a call to the setpwent function will reset the reading of password entries to the beginning of the /etc/passwd file.

Return Value

If the function is successful, the function will return a pointer to a TPasswordRecord structure that represents a user on the system. If there are no more entries to read in the file, or the function results in an error, the function will return nil.

See Also

setpwent, endpwent

Example

Listing 4.37 - Using the getpwent function

```

procedure TfrmUsers.ListUsers(AStrings: TStrings);
var
  usr: PPasswordRecord;
begin
  AStrings.Clear;
  //Reset the /etc/passwd file read position
  setpwent;
  try
    //Read all the entries
    usr := getpwent;
    while usr <> nil do
      begin
        AStrings.Add(usr^.pw_name + '(' +
          IntToStr(usr^.pw_uid)+' ');
        usr := getpwent;
      end;
    finally
      endpwent;
    end;
  end;
end;

```

getpwnam *LibC.pas*

Syntax

```

function getpwnam(
  Name: PChar
):PPasswordRecord;

```

Description

The getpwnam function will return the details about a user from the /etc/passwd file, when only the username is known. The function will return the details in the form of a pointer to a TPasswordRecord that represents the stored information in the /etc/passwd file for that user.

```

type
  passwd = record
    pw_name: PChar;           // Username
    pw_passwd: PChar;         // Password
    pw_uid: __uid_t;          // User ID
    pw_gid: __gid_t;          // Group ID
    pw_gecos: PChar;          // Real name
    pw_dir: PChar;            // Home directory
    pw_shell: PChar;          // Shell program
  end;
  TPasswordRecord = passwd;
  PPasswordRecord = ^TPasswordRecord;

```

Parameters

Name: This parameter is the username as a null-terminated string.

Return Value

If the username passed to the Name parameter is found in the `/etc/passwd` file, the function will return a pointer to a `TPasswordRecord` which represents the details of the user. If the username is not found in the `/etc/passwd` file or an error occurred while using the function, it will return `nil`.

See Also

`setpwent`, `endpwent`, `getpwent`, `getpwuid`

Example

Listing 4.38 - Using the `getpwnam` function

```
procedure TfrmUsers.FindUserByName;
var
  usr: TPasswordRecord;
  strUserName: string;
begin
  mmSearchResults.Lines.Clear;
  strUserName := edSearchValue.Text;
  usr := getpwnam(PChar(strUserName));
  if usr = nil then
    mmSearchResults.Lines.Add('No User with that name was found.')
  else begin
    mmSearchResults.Lines.Add('User ID: '+inttostr(usr^.pw_uid));
    mmSearchResults.Lines.Add('User Name: '+usr^.pw_name);
    mmSearchResults.Lines.Add('User Password: '+usr^.pw_passwd);
    mmSearchResults.Lines.Add('Group ID: '+inttostr(usr^.pw_gid));
    mmSearchResults.Lines.Add('User Directory: '+usr^.pw_dir);
    mmSearchResults.Lines.Add('User Shell: '+usr^.pw_shell);
  end;
end;
```

getpwuid *LibC.pas*

Syntax

```
function getpwuid(
  uid: __uid_t
):TPasswordRecord;
```

Description

The `getpwuid` function will return the details about a particular user from the `/etc/passwd` file when only the ID of the user is known. The information about the user will be returned as a pointer to a `TPasswordRecord` that represents the user.

```
type
  passwd = record
    pw_name: PChar;           // Username
    pw_passwd: PChar;         // Password
    pw_uid: __uid_t;          // User ID
    pw_gid: __gid_t;          // Group ID
    pw_gecos: PChar;          // Real name
```

```

    pw_dir: PChar;           // Home directory
    pw_shell: PChar;        // Shell program.
end;
TPasswordRecord = passwd;
PPasswordRecord = ^TPasswordRecord;

```

Parameters

uid: The uid parameter is an integer value that can be matched to a record in the `/etc/passwd` file.

Return Value

If the user identifier passed to the uid parameter is valid, the function will return a pointer to a `TPasswordRecord` which signifies the details of the user. If the uid parameter does not represent a valid user, or an error occurred while using the function, the function will return `nil`.

See Also

`setpwent`, `endpwent`, `getpwent`, `getpwnam`

Example

Listing 4.39 - Using the `getpwuid` function

```

procedure TfrmUsers.FindUserByID;
var
    usr: PPasswordRecord;
begin
    mmSearchResults.Lines.Clear;
    usr := getpwuid(strtoint(edSearchValue.Text));
    if usr = nil then
        mmSearchResults.Lines.Add('No User with that ID was found.')
    else begin
        mmSearchResults.Lines.Add('User ID: '+inttostr(usr^.pw_uid));
        mmSearchResults.Lines.Add('User Name: '+usr^.pw_name);
        mmSearchResults.Lines.Add('User Password: '+usr^.pw_passwd);
        mmSearchResults.Lines.Add('Group ID: '+inttostr(usr^.pw_gid));
        mmSearchResults.Lines.Add('User Directory: '+usr^.pw_dir);
        mmSearchResults.Lines.Add('User Shell: '+usr^.pw_shell);
    end;
end;

```

getuid *LibC.pas*

Syntax

```
function getuid: __uid_t;
```

Description

This function returns the real user ID of the process, that is, the user who is currently executing the process.

Return Value

The function always returns the user ID of the user who is running the application. This return value should be passed into `getpwuid` to obtain more details about the user.

See Also

`geteuid`, `getgid`, `getpwuid`

Example

See Listing 4.27 - Displaying user information about a process.

group_member LibC.pas**Syntax**

```
function group_member(
  GroupID: __gid_t;
):Integer;
```

Description

This function determines whether the current process is in the group specified by the group parameter. This function can be used to determine permissions for a particular file, directory, or resource.

Parameters

GroupID: This is the group ID to examine to see if the process is in the group.

Return Value

If the process is in the group, the function returns 1. If the process is not in the group, the function returns 0.

See Also

`getgroups`, `getgrent`

Example**Listing 4.40 - Using the group_member function**

```
procedure TfrmLinuxGroups.ReadInGroups;
var
  grp: PGroup;
  DoesProcessHavePermissionsOfGroup: boolean;
  intIndex: integer;
begin
  //Read in the list of groups from the system and check
  //the groups the current process is a member of
  grp := getgrent;
  while grp <> nil do
    begin
      //Display the information about the group
      intIndex := clbGroups.Items.Add(grp^.gr_name);
```

```

DoesProcessHavePermissionsOfGroup :=
  group_member(grp^.gr_gid) = 1;
c1bGroups.Checked[intIndex] := DoesProcessHavePermissionsOfGroup;

  grp := getgrnt;
end;
end;

```

putenv LibC.pas

Syntax

```

function putenv(
  Str: PChar;
):Integer;

```

Description

The putenv function sets an environment name/value pair in the current environment.

This function only pertains to the current process and any child processes that are created from the application using the fork or exec functions.

Parameters

Str: Str sets an environment variable string in the form of name=value. If the named environment variable exists in the environment, it will be changed to the new value. If it does not, it will be added to the environment.

If the Str parameter only holds the name value in the form name, the environment variable will be removed.

Return Value

The function returns 0 if successful or -1 if the environment variable is not written in the case of name=value or deleted in the case of name.

See Also

getenv, setenv, clearenv, __secure_getenv

Example

Listing 4.4I - Using the putenv function

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  //Store the environment value and then create a child process
  //that can use the settings value
  if putenv('SimpleDBAppSettings=/root/dblocation.ini') = 0 then
  begin
    if fork = 0 then
    begin
      //In the child process
      RunChildActions;
      Exit;
    end;
  end else begin

```

```

        MessageDlg('The settings file could not be set.', mtInformation,
        [mbOk], 0);
    end;
end;

```

setegid LibC.pas

Syntax

```

function setegid(
  GroupID: __gid_t;
):Integer;

```

Description

The setegid function sets the effective group ID of the calling process. This alters the access levels of the process, so that when accessing files from within the application, the process will take into account the permission of the new effective group.

Parameters

GroupID: This is the new group ID that will be used as the effective group ID for the process.

Return Value

The function returns 0 when successful. If the function fails, it returns -1.

See Also

seteuid

Example

Listing 4.42 - Using the setegid function

```

procedure TfrmLinuxGroups.btnSetEffectiveGroupClick(Sender: TObject);
var
  intGroupID: integer;
begin
  //Get the Group ID from the CheckListBox
  if clbGroups.ItemIndex = -1 then
    Exit;
  intGroupID := integer(clbGroups.Items.Objects[clbGroups.ItemIndex]);

  //Now set the new effective group ID
  setegid(intGroupID);

  //Refresh the Users
  RefreshGroupMembershipDisplay;
end;

```

setenv LibC.pas**Syntax**

```
function setenv(
  Name: PChar;
  const Value: PChar;
  Replace: Integer
):Integer;
```

Description

The setenv function sets an environment name/value pair in the current environment.

This function only pertains to the current process and any child processes that are created from the application using the fork or exec functions.

Parameters

Name: Name refers to the name of the environment value in the name/value pair.

Value: This holds a reference to a string containing the value of the environment variables to set.

Replace: The Replace parameter will only be used if the environment variable in the Name parameter refers to an existing environment variable and determines if the environment variable should be overwritten. When Replace is a non-zero value, the named environment value exists and will be replaced by the new value. When it is 0, no change will be made to the environment.

Return Value

The function returns 0 if successful or -1 if the environment variable is not written.

See Also

getenv, putenv, clearenv, __secure_getenv

Example**Listing 4.43 - Using the setenv function**

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //Store the environment value and then create a child process
  //that can use the settings value
  if setenv('SimpleDBAppSettings', '/root/dblocation.ini',
    -1) = 0 then
  begin
    if fork = 0 then
    begin
      //In the child process
      RunChildActions;
      Exit;
    end;
  end else begin
    MessageDlg('The settings file could not be set.', mtInformation,
```

```

        [mb0k], 0);
    end;
end;

```

seteuid LibC.pas

Syntax

```

function seteuid(
  UID: __uid_t;
):Integer;

```

Description

This function will set the effective user ID of the application to the user ID specified in the UID parameter. This function can only be executed by the superuser of the system. A function such as a Web server may use this function internally when a user logs into a Web page and may translate the username and password to a UID and then impersonate the access privileges of that user.

Parameters

UID: This is the user ID that the process will have set as its effective user ID. This value will be used in determining permissions to specific user privileges.

Return Value

If the function was successful, then 0 is returned; otherwise, -1 is returned by the function and a call to `errno` will determine the result of the function.

See Also

`getuid`, `geteuid`

Example

Listing 4.44 - Using the seteuid function

```

procedure TfrmLinuxGroups.ReadInUsers;
var
  usr: PPasswordRecord;
begin
  //Read in the list of users on the system
  usr := getpwent;
  while usr <> nil do
    begin
      //Display the information about the user
      clbUsers.Items.AddObject(usr^.pw_name,
        TObject(usr^.pw_uid));

      usr := getpwent;
    end;

    RefreshUserDisplay;
  end;

```

```

procedure TfrmLinuxGroups.RefreshUserDisplay;
begin
    lblEffectiveUserID.Caption := UserIDToUserName(geteuid);
end;

procedure TfrmLinuxGroups.btnSetEffectiveUserIDClick(Sender: TObject);
var
    intUserID: integer;
begin
    //Get the group ID from the CheckListBox
    if clbUsers.ItemIndex = -1 then
        Exit;
    intUserID := integer(clbUsers.Items.Objects[clbUsers.ItemIndex]);

    //Now set the new effective group ID
    if seteuid(intUserID) = -1 then
        MessageDlg('You cannot set the effective user ID. It is possible that '+
            'the current process's permissions do not allow this.',
            mtWarning, [mbOk], 0);

    //Refresh the Users
    RefreshUserDisplay;
end;

```

setgid LibC.pas

Syntax

```

function setgid(
    GroupID: __gid_t;
):Integer;

```

Description

The setgid function is used to change the effective group of the process to the user ID specified by the UID parameter. When this command is executed by the superuser account, then the real group ID will also be set to the group ID denoted by the GroupID parameter.

Parameters

GroupID: This is the group ID to be used as the new effective group ID.

Return Value

When successful, the function returns 0. If the function fails, it returns -1. When this function results in an error it is normally due to the process not having enough permission to successfully change the group of the process. Normally only those processes running as root have these permissions.

See Also

setgid, setuid, seteuid, getuid, getgid, getegid, geteuid

*Example***Listing 4.45 - Using the setgid function**

```

procedure TfrmLinuxGroups.btnSetEffectiveGroupClick(Sender: TObject);
var
    intGroupID: integer;
begin
    //Get the Group ID from the CheckListBox
    if clbGroups.ItemIndex = -1 then
        Exit;
    intGroupID := integer(clbGroups.Items.Objects[clbGroups.ItemIndex]);

    //Now set the new effective group ID
    setgid(intGroupID);

    //Refresh the Users
    RefreshGroupMembershipDisplay;
end;

```

setgrent LibC.pas*Syntax*

```
procedure setgrent;
```

Description

The setgrent function opens the Linux group file /etc/group so that information can be read from the group file using the getgrent function. The main purpose of this function is to rewind the reading position to the start of the /etc/group file.

See Also

getgrent, endgrent, getgrgid, getgrnam

Example

See Listing 4.29 - Using the getgrent function.

setpgid LibC.pas*Syntax*

```

function setpgid(
    ProcessID: __pid_t;
    ProcessGrpID: __pid_t;
):Integer;

```

Description

This function will set the process group ID of a particular process.

Parameters

ProcessID: This is the process ID whose process group will be set. If this value is 0, the function will set the process group ID of the current process.

ProcessGrpID: This is the group ID to set to the process to. When the value is set to 0, the value will be the same as the current process identifier.

Return Value

When successful, this function returns 0. If the function fails, it returns -1.

See Also

setpgrp

Example

Listing 4.46 - Using the setpgid function

```
procedure TfrmMain.btnSetNewProcessGroupClick(Sender: TObject);
begin
    //Set the new process group ID.
    setpgid(0, 0);
end;
```

setpgrp LibC.pas

Syntax

```
function setpgrp:Integer;
```

Description

The setpgrp function sets the process group ID to the same value of the process ID, effectively creating a new process group and setting the current process as the process group leader. This function call is the equivalent of calling setpgid(0, 0);

Return Value

When successful, the setpgrp function returns 0. If the function fails for any reason, it returns -1.

See Also

setpgid, getpgrp

Example

Listing 4.47 - Using the setpgrp function

```
procedure TfrmMain.btnSetNewProcessGroupClick(Sender: TObject);
begin
    //Set the new process group id.
    setpgrp;
end;
```


setpwent LibC.pas*Syntax*

```
procedure setpwent;
```

Description

The setpwent function opens the Linux password file /etc/passwd so that information can be read from the password file using the getpwent function. The main purpose of this function is to rewind the reading position to the start of the /etc/passwd file.

See Also

getpwent, endpwent

Example

See Listing 4.37 - Using the getpwent function.

setregid LibC.pas*Syntax*

```
function setregid(
  RGID: __gid_t;
  EGID: __gid_t;
):Integer;
```

Description

This function sets the real and effective group IDs in one call.

Parameters

RGID: This is the real group ID that the process will be set to after a successful function call.

EGID: This is the effective group ID that the process will be set to after a successful function call.

Return Value

When successful, the function returns 0. If there is any error, the function returns -1. The function will normally fail if the user that is running the process does not have sufficient permission to change the group IDs.

See Also

setreuid

*Example***Listing 4.48 - Using the setregid function**

```

procedure TfrmLinuxGroups.btnSetRealAndEffectiveGroupIDsClick(
  Sender: TObject);
var
  intGroupID: integer;
begin
  //Get the Group ID from the CheckListBox
  if clbGroups.ItemIndex = -1 then
    Exit;
  intGroupID := integer(clbGroups.Items.Objects[clbGroups.ItemIndex]);

  //Now set the new effective group ID
  if setregid(intGroupID, intGroupID) = -1 then
    MessageDlg('You cannot set the real & effective group ID. It is possible that '+'
      'the current process's permissions do not allow this.',
      mtWarning, [mbOk], 0);

  //Refresh the Users
  RefreshGroupMembershipDisplay;
end;

```

setreuid LibC.pas*Syntax*

```

function setreuid(
  RUID: __uid_t;
  EUID: __uid_t;
):Integer;

```

Description

The setreuid function sets the real and effective user IDs in a single call.

Parameters

RUID: This is the real user ID that the process will be set to after a successful function call.

EUID: This is the effective user ID that the process will be set to after a successful function call.

Return Value

When successful the function returns 0, and if there is any error the function returns -1.

The function will normally fail if the user that is running the process does not have sufficient permission to change the user IDs.

See Also

setregid, seteuid, setuid, getuid, geteuid

*Example***Listing 4.49 - Using the setreuid function**

```

procedure TfrmLinuxGroups.btnSetRealAndEffectiveUserIDsClick(
  Sender: TObject);
var
  intUserID: integer;
begin
  //Get the group ID from the CheckListBox
  if clbUsers.ItemIndex = -1 then
    Exit;
  intUserID := integer(clbUsers.Items.Objects[clbUsers.ItemIndex]);

  //Now set the new effective group ID
  if setreuid(intUserID, intUserID) = -1 then
    MessageDlg('You cannot set the real and effective user ID. It is possible that '+
      'the current process's permissions do not allow this.',
      mtWarning, [mbOk], 0);

  //Refresh the Users
  RefreshUserDisplay;
end;

```

setuid LibC.pas*Syntax*

```

function setuid(
  UID: __uid_t;
):Integer;

```

Description

The setuid function is used to change the effective user of the process to the user ID specified by the UID parameter. When this command is executed by the superuser account, then the real user ID will also be set to the user ID denoted by the UID parameter.

Parameters

UID: This is the effective user ID of the process after a successful function call.

Return Value

When successful, this function returns 0. If the function fails, it returns -1. A failure is normally a result of insufficient permissions of the current process.

See Also

seteuid, setegid, getuid

Example

Listing 4.50 - Using the setuid function

```

procedure TfrmLinuxGroups.btnSetEffectiveUserIDClick(Sender: TObject);
var
    intUserID: integer;
begin
    //Get the Group ID from the CheckListBox
    if clbUsers.ItemIndex = -1 then
        Exit;
    intUserID := integer(clbUsers.Items.Objects[clbUsers.ItemIndex]);

    //Now set the new effective group ID
    if setuid(intUserID) = -1 then
        MessageDlg('You cannot set the effective user ID. It is possible that '+
            'the current process's permissions do not allow this.',
            mtWarning, [mbOk], 0);

    //Refresh the Users
    RefreshUserDisplay;
end;

```

system LibC.pas

Syntax

```

function system(
    Command: PChar
):Integer;

```

Description

The system function executes a Linux command passed into the Command parameter. Often this is a simple Linux command, such as ls, or it could be a completely different application. The function returns after the command has been completed unless the & character is appended to the command, in which case the function will return immediately.

Parameters

Command: This is the command that is to be executed. The command that is executed should at least be able to be completed from the Linux shell. If you want the function to return immediately, append the command with the ampersand (&) character. Passing a nil value to this parameter can also be used to determine if a shell is available.

Return Value

This function returns -1 if the function call was unsuccessful, 127 if the /bin/sh shell that executes the command could not be executed, and 0 if the function was successful. If nil was passed to the Command parameter, then a return of 0 indicates that a shell is installed on the system.

Example

See Listing 4.8 - Using the LibC.system command.

unsetenv LibC.pas**Syntax**

```
procedure unsetenv(
  Name: PChar;
);
```

Description

The `unsetenv` function removes the environment variable passed into the `Name` parameter from the process's environment.

Parameters

`Name`: This is the Name of the environment name/value pair to remove.

See Also

`getenv`, `setenv`, `putenv`, `clearenv`

Example

See Listing 4.2 - Source code to environment application.

vfork LibC.pas**Syntax**

```
function vfork: __pid_t;
```

Description

The `vfork` function is similar to the `fork` function call in that it will make a duplicate of the current process and return twice. The difference between the `fork` and `vfork` functions is that `vfork` will not copy the original process's memory. Instead, it will use the original process's memory until the copy of the original process either exits or uses one of the `exec` functions to replace a particular process.

Return Value

If the function is successful, the original process will return the new process ID of the child process and the child process will return 0. If the function fails, the function will return `-1` in the parent process. The function will normally only fail if there is insufficient memory to create a new process.

See Also

`fork`, `exec`

Example

Listing 4.51 - Using the vfork function

```

procedure TfrmVforkDemo.btnRun_xcalcClick(Sender: TObject);
var
  vforkRes: integer;
begin
  //This will simply fork and then
  //run the xcalc utility immediately if it's the child process.
  vforkRes := vfork;
  if vforkRes = 0 then
    execlp('xcalc', 'xcalc', nil)
  end;
end;

```

wait LibC.pas

Syntax

```

function wait(
  __stat_loc: PInteger;
):pid_t;

```

Description

The wait function waits for any of a process's child processes to terminate and returns the details about how the child process terminated. When calling this function, the calling process will not return until a child has terminated.

Parameters

__stat_loc: This is the status of the child process that terminated. This result should be passed to one of the functions listed in Table 4.2. This value can also be nil, which will result in no status being returned.

Return Value

When this function call is successful, the function returns the process ID of the child that was terminated.

See Also

wait3, wait4, waitpid

Example

Listing 4.52 - Using the wait function

```

program WaitingForAnyChildDemo;

{$APPTYPE CONSOLE}

uses
  Libc;

```

```

var
    CloneProcessID: integer;
    Status: integer;
    ProcessThatTerminated: integer;

begin
    writeln(getpid, ' (Parent): Starting the Cloning Process.');
```

CloneProcessID := fork;

```

    case CloneProcessID of
        0: begin
            //We are the cloned application
            writeln(getpid, ' (Child): We are the Cloned Application.');
```

//Sending the SIGKILL signal here is used to demonstrate
//examining the Status from a terminated child and would
//not be used in normal applications.
__raise(SIGKILL);

```

            WriteLn(getpid, ' (Child): Finished.');
```

end;

```

        -1: begin
            //There was an error in the forking process
            writeln(getpid, ' (Parent): An error occurred while forking.');
```

end;

```

    else
        begin
            //The fork was successful
            //Let's wait until the child has successfully done its thing.
            writeln(getpid, ' (Parent): Waiting for any child process to finish');
```

ProcessThatTerminated := wait(@Status);
writeln('The process that terminated was ', ProcessThatTerminated);

```

            //Write the reason the child process ended.
            write(getpid, ' (Parent): ');
            if WIFEXITED(Status) then
                writeln('The process ended normally.')
```

else if WEXITSTATUS(Status) <> 0 then
writeln('The child was exited.')

```

            else if WTERMSIG(Status) > 0 then
                writeln('The child retrieved the terminating signal')
```

else if WSTOPSIG(Status) > 0 then
writeln('A signal caused the process to stop')

```

            else if WIFSIGNALED(Status) then
                writeln('A signal caused the process to stop')
```

else if WIFSTOPPED(Status) then
writeln('The child is stopped.')

```

            else if WCOREDUMP(Status) then
                writeln('The process dumped its core.');
```

writeln(getpid, ' (Parent): The child has finished.');

```

        end;
    end;
end.
```

wait3 LibC.pas*Syntax*

Note: This function is incorrect in the LibC.pas unit that ships with Kylix. The following function declaration will correct the problem if included in a unit. The correct version of the function is included in the LibcHelperFunctions.pas file on the companion CD.

```
function wait3(
  __stat_loc: PInteger;
  __options: Integer;
  __usage: PRUsage
):pid_t; cdecl; external libcmodule name 'wait3';
```

Description

The wait3 function waits for any of a process's child processes to terminate, returns the details about how it terminated, and will return information about the resource usage of the process.

Parameters

__stat_loc: This value is either a nil pointer or a pointer to an integer where the details about the status of the child will be returned. This result should be passed to one of the functions listed in Table 4.2.

__options: This is the OR'ed value of either WNOHANG or WUNTRACED. The WNOHANG option will return immediately if no child has finished. The WUNTRACED option means that the function will return if processes are simply stopped or have an unknown status.

__usage: This value is a pointer to a TRUsage record structure that defines the resource of the child process that had exited.

Return Value

When successful, the function returns the process ID of the process that exited. If the WNOHANG option was used and there were no children that exited, the function returns 0. If the function fails, it returns -1.

See Also

wait, waitpid, wait4

*Example***Listing 4.53 - Using the wait3 function**

```
procedure TfrmMain.btnMakeANewCopyClick(Sender: TObject);
var
  intRes: integer;
```



```

    ApplicationName: array[0..500] of char;
begin
    StrPCopy(ApplicationName, ParamStr(0));
    intRes := vfork;
    if intRes = 0 then
        execl(ApplicationName, ApplicationName, nil)
    else
        LastProcessCreated := intRes;
end;

procedure TfrmMain.btnWaitForProcessClick(Sender: TObject);
var
    Status: integer;
    Usage: TRUsage;
    waitResult: integer;
begin
    //wait for the process
    if rbUseWait3.Checked then
        waitResult := wait3(@Status, 0, @Usage)
    else
        waitResult := wait4(LastProcessCreated, @Status, 0, @Usage);

    if waitResult <> -1 then
    begin
        //Display the Resource information
        with mmProcessGroupInfo.Lines do
        begin
            Add('Process ID '+inttostr(waitResult)+' ended.');
            Add('Resource Information.');
            Add('User Time (Seconds/NanoSeconds: '+
                inttostr(Usage.ru_utime.tv_sec)+'/'+inttostr(Usage.ru_utime.tv_usec));
            Add('System Time (Seconds/NanoSeconds: '+
                inttostr(Usage.ru_stime.tv_sec)+'/'+inttostr(Usage.ru_stime.tv_usec));
            Add('Maximum resident set size (kilobytes): '+inttostr(Usage.ru_maxrss));
            Add('Segment Sharing with other processes (kb/sec): '+inttostr(Usage.ru_ixrss));
            Add('Data Segment Memory Used (kb/sec): '+inttostr(Usage.ru_idrss));
            Add('No of soft page faults: '+inttostr(Usage.ru_minflt));
            Add('No of hard page faults: '+inttostr(Usage.ru_majflt));
            Add('No of times the process was swapped out of physical memory: '+
                inttostr(Usage.ru_nswap));
            Add('No of input operations with the file system: '+
                inttostr(Usage.ru_inblock));
            Add('No of output operations with the file system: '+
                inttostr(Usage.ru_oublock));
            Add('No of IPC messages sent: '+inttostr(Usage.ru_msgsnd));
            Add('No of IPC messages received: '+inttostr(Usage.ru_msgrcv));
            Add('No of signals delivered: '+inttostr(Usage.ru_signals));
            Add('No of voluntary context switches: '+inttostr(Usage.ru_nvcsw));
            Add('No of involuntary context switches: '+inttostr(Usage.ru_nivcsw));
        end;
    end else
        mmProcessGroupInfo.Lines.Add('There was a problem waiting for the process.');
end;

```

wait4 LibC.pas

Note: This function is incorrect in the LibC.pas unit that ships with Kylix. The following function declaration will correct the problem if included in a unit. The correct version of the function is included in the LibcHelperFunctions.pas file on the companion CD.

Syntax

```
function wait4(
  __pid: pid_t;
  __stat_loc: PInteger;
  __options: Integer;
  __usage: PRUsage
):pid_t; cdecl; external libcmodule name 'wait4';
```

Description

The wait4 function will wait until a particular process has exited, any child process has exited, or any child process in a specific process group has exited. The exit code of the child process that terminated is returned in the __stat_loc parameter. The function also includes process information about the terminated child process.

Parameters

__pid: This parameter defines which process should be waited for. If this value is a positive integer, the process will wait for that particular process to finish. When this parameter is -1, the function will wait for any child process to finish. When this parameter is 0, the function will wait until any process in the same group as the calling process has finished. When this value is less than -1, the function will wait until a child process with the absolute value of the __pid parameter is finished.

__stat_loc: This is the status of the child process that terminated. This result should be passed to one of the functions listed in Table 4.2. This value can also be nil, which will result in no status being returned.

__options: This is the OR'ed value of either WNOHANG or WUNTRACED. The WNOHANG option will return immediately if no child has finished. The WUNTRACED option means that the function will return if processes are simply stopped or have an unknown status.

__usage: This value is a pointer to a TRUsage record structure that defines the resource of the child process that had exited.

Return Value

When successful, the function returns the process ID of the process that exited. If the WNOHANG option was used and there were no children that exited, the function returns 0. If the function fails, then it returns -1.

See Also

wait, wait3, waitpid

Example

See Listing 4.53 - Using the wait3 function.

waitpid LibC.pas*Syntax*

```
function waitpid(
  __pid: pid_t;
  __stat_loc: PInteger;
  __options: Integer
):pid_t;
```

Description

The waitpid function will wait until a particular process has exited, any child process has exited, or any child process in a specific process group has exited. The exit code of the child process that terminated is returned in the `__stat_loc` parameter.

Parameters

`__pid`: This parameter defines which process should be waited for. If this value is a positive integer, the process will wait for that particular process to finish. When this parameter is `-1`, the function will wait for any child process to finish. When this parameter is `0`, the function will wait until any process in the same group as the calling process has finished. When this value is less than `-1`, the function will wait until a child process with the absolute value of the `__pid` parameter is finished.

`__stat_loc`: This is the status of the child process that terminated. This result should be passed to one of the functions listed in Table 4.2. This value can also be `nil`, which will result in no status being returned.

`__options`: This is the OR'ed value of either `WNOHANG` or `WUNTRACED`. The `WNOHANG` option will return immediately if no child process has finished. The `WUNTRACED` option means that the function will return if processes are simply stopped or have an unknown status.

Return Value

When successful, the function returns the process ID of the process that exited. If the `WNOHANG` option was used and there were no children that exited, the function returns `0`. If the function fails, it returns `-1`.

See Also

wait, wait3, wait4

*Example***Listing 4.54 - Using the waitpid function**

```

program waitpidExample;

{$APPTYPE CONSOLE}

uses
  Libc;

var
  ClonedProcess: integer;
  ChildStatus: integer;
  WaitResult: integer;

begin
  writeln('About to fork the process.');
```

```

  ClonedProcess := fork;
  if ClonedProcess = 0 then
    begin
      //We are the child. Let's sleep on it
      writeln('The child process is about to sleep');
```

```

      _sleep(5);
      writeln('The child process is awake.');
```

```

    end else if ClonedProcess > 0 then begin
      writeln('The fork was successful.');
```

```

      writeln('Waiting on the child process to exit.');
```

```

      WaitResult := waitpid(ClonedProcess, @ChildStatus, 0);
      if WaitResult <> -1 then
        writeln('The child ', WaitResult, ' has exited.')
```

```

      else
        perror('There was a problem waiting.');
```

```

    end else begin
      //We have an error
      perror('There was an error forking.');
```

```

    end;
end.
```

A Practical Look at Processes — A Server Health Check

This chapter's practical use for processes is creating a daemon that allows you to check if a group of servers have not crashed. This is quite a simple daemon that checks if a server is able to be connected to. It does this by periodically pinging the servers that need to be connected to. If the connection could not be made to any of the servers, an e-mail is sent to a specific e-mail address, announcing the inability to connect to the server. The Internet functionality is done in these examples by using the Indy components.

The example also demonstrates some good practices of creating daemons for Linux. Windows users may have come across a scenario where you install or reconfigure a Windows service, but the change is not activated until the service is restarted or, even worse, the machine is rebooted.

To combat the above scenario under Linux, the configuration file that holds the information about what servers to test and which e-mail addresses to send messages to is read in

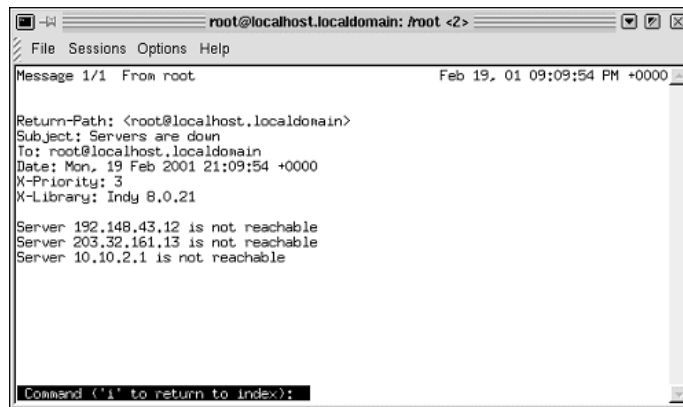
every time the server health checks are performed. This increases the usability of the daemon by allowing you to update the servers without a restart.

Another good practice for a daemon is to take up as little resources as it can. To combat this the daemon only performs the check at regular intervals which is set to every 10 minutes. Once the check is complete, the service hibernates until it needs to perform the next check. This minimizes the use of resources of the application so that your daemon does not take too much processing time.

The daemon also logs all messages to its own log file. A server that runs indefinitely should write its log file entries to individual log files each day so that the log files can be removed periodically by the Linux machine's system administrator.

Finally, you should be able to use the daemon to test any NetCLX applications that are running on Apache. If you are running a 24/7 application, you should be made aware of any server downtime as soon as possible. A good use of this application is to send the e-mail to an e-mail address that forwards any messages to a mobile phone as an SMS message. In this case, the server can have as little downtime as possible with a quick response time when any problems occur.

Figure 4.16 - An e-mail coming from the server's health check daemon



Listing 4.55 - Using the server health check daemon

```

program ServerHealthCheck;

uses
  Classes, SysUtils, Libc, IdBaseComponent, IdComponent, IdRawBase,
  IdRawClient, IdIcmpClient, IdSMTP;

{$APPTYPE CONSOLE}

const
  SettingsFile = '/etc/ServerHealthCheck/settings';
  IPListFile = '/etc/ServerHealthCheck/IPList';
  ErrorLogDirectory = '/etc/ServerHealthCheck/Errors/';

var
  pid, sid: integer;
  Done: boolean;
  ServersToCheck: TStringList;
  
```

```

Settings: TStringList;
ServersNotRunning: TStringList;
EmailAddressToSendTo: string;
EmailAddressToSendFrom: string;
EmailServer: string;
counter: integer;
CheckEveryXMinutes: integer;
StartCheckTime, EndCheckTime: TDateTime;
AmountOfTimeToWait: TDateTime;
SMTP: TIdSMTP;

function IsServerRunning(IPAddress: string): boolean;
var
    IdIcmpClient: TIdIcmpClient;
begin
    //This function requires the IdBaseComponent, IdComponent, IdRawBase, IdRawClient,
    //and IdIcmpClient units
    IdIcmpClient := TIdIcmpClient.Create(nil);
    try
        try
            //See if the Web sites are still running
            IdIcmpClient.Host := IPAddress;
            IdIcmpClient.Ping;
            Result := true;
        except
            Result := false;
        end;
    finally
        IdIcmpClient.Free;
    end;
end;

procedure LogErrorMessage(sMessage: string);
var
    f: TextFile;
    LogFile: string;
begin
    //Here for our customized daemon, we are using our own log file.
    LogFile := ErrorLogDirectory+'errors'+
        FormatDateTime('mmddyyyy', Now)+'.log';

    AssignFile(f, LogFile);
    try
        if FileExists(LogFile) then
            Append(f)
        else
            Rewrite(f);
        writeln(f, FormatDateTime('ddd mm/dd/yyyy hh:nn:ss', Now)+' : '+sMessage);
    finally
        CloseFile(f);
    end;
end;

function GetSecondsFromTTime(dt: TDateTime): integer;
var
    h, n, s, ms: word;
begin

```

```

    DecodeTime(dt, h, n, s, ms);
    Result := (h * 60 * 60) + n * 60 + s;
end;

begin
    //Fork the process, so we can have an orphan daemon
    pid := fork;

    //See if there was a problem forking
    if (pid = -1) or (pid = 127) then
    begin
        //There was an error forking
        LogErrorMessage('fork creation');
        Exit;
    end;

    if pid > 0 then
    begin
        //We are the original process, so we need
        //to terminate the parent process
        Exit;
    end;

    //If the application gets to here it means
    //that this process will be the one that lives

    //Start a new Process Session
    sid := setsid;
    if sid < 0 then
    begin
        LogErrorMessage('Set Session ID Failed');
        Exit;
    end;

    //Change the directory to the '/' dir.
    //This ensures there is no lag time which
    //may be caused by trying to access devices such as /mnt/floppy
    __chdir('/');

    //Set the umask for the creation of any new files
    umask(0);

    //Close all the streams
    __close(STDIN_FILENO);
    __close(STDOUT_FILENO);
    __close(STDERR_FILENO);

    //Now here is where all the work for the
    //daemon occurs.
    LogErrorMessage('Daemon ServerHealthCheck started');

    //Set the default time to wait
    CheckEveryXMinutes := 15;

    Done := false;
    while not Done do
    begin

```

```

//Get the starting time, so that we can make sure that this
//process occurs every X minutes
StartCheckTime := Now;

//Load the settings for this daemon. We do this every time so that the
//daemon can receive changes without having to restart
if FileExists(SettingsFile) then
begin
    Settings := TStringList.Create;
    try
        Settings.LoadFromFile(SettingsFile);
        CheckEveryXMinutes := strtointDef(Settings.Values['EveryXMinutes'],
            CheckEveryXMinutes);
        EmailServer := Settings.Values['EmailServer'];
        EmailAddressToSendTo := Settings.Values['EmailAddressTo'];
        EmailAddressToSendFrom := Settings.Values['EmailAddressFrom'];
        EmailServer := Settings.Values['EmailServer'];
    finally
        Settings.Free;
    end;
end;

//Load the IP Address List.
ServersToCheck := TStringList.Create;
try
    if FileExists(IPListFile) then
        ServersToCheck.LoadFromFile(IPListFile);

ServersNotRunning := TStringList.Create;
try
    //Check the health of all the servers
    for counter := 0 to ServersToCheck.Count - 1 do
    begin
        if not IsServerRunning(ServersToCheck[counter]) then
        begin
            LogErrorMessage('Server '+ServersToCheck[counter]+ ' is not reachable');
            ServersNotRunning.Add('Server '+ServersToCheck[counter]+ ' is not
                reachable')
        end;
    end;

    //Email the message to someone that the server is not reachable
    //only if there are servers down and the email settings are
    //ok
    if (ServersNotRunning.Count > 0) and (EmailAddressToSendTo <> '')
        and (EmailServer <> '') then
    begin
        //Let's send the email to the happy support person
        //who is probably asleep and the time is
        //4 in the morning....
        SMTP := TIdSMTP.Create(nil);
        try
            SMTP.QuickSend(EmailServer, 'Servers are down',
                EmailAddressToSendTo, EmailAddressToSendFrom,
                ServersNotRunning.Text);
        except
            //The Email message could not be sent.

```



```

        LogErrorMessage('Error: Email could not be sent');
    end;
end;
finally
    ServersNotRunning.Free;
end;
finally
    ServersToCheck.Free;
end;

//Now we delay the application a particular amount so that the
//daemon will check itself every 1 minute
EndCheckTime := Now;
AmountOfTimeToWait := (CheckEveryXMinutes * EncodeTime(0, 1, 0, 0)) -
    (EndCheckTime - StartCheckTime);

    sleep(GetSecondsFromTTime(AmountOfTimeToWait));
end;
end.
```

Conclusion

It is impossible to ignore processes under Linux. They are the building blocks of any serious, or not so serious, application. From looking at the information about processes to building robust daemons, the techniques you've learned will help as you use process information to your advantage.

Now that you've learned to use processes, you can learn the best techniques for communicating between processes, also known as interprocess communication (IPC).



Interprocess Communication (IPC)

Introduction

This chapter looks at what is involved in communicating between running processes on a single Linux machine. Interprocess communication is an important part of setting up reliable server applications. As we saw in Chapter 4, processes are an important way of setting up applications that require real-time performance with mission-critical reliability.

This chapter discusses some of the most widely used Linux API functions for communication between processes on a single machine such as signals, pipes, named pipes, message queues, and shared memory. Although not all of these methods will fit the way your application works, by getting an understanding of the best methods of process communication, you can design and architect your applications more reliably.

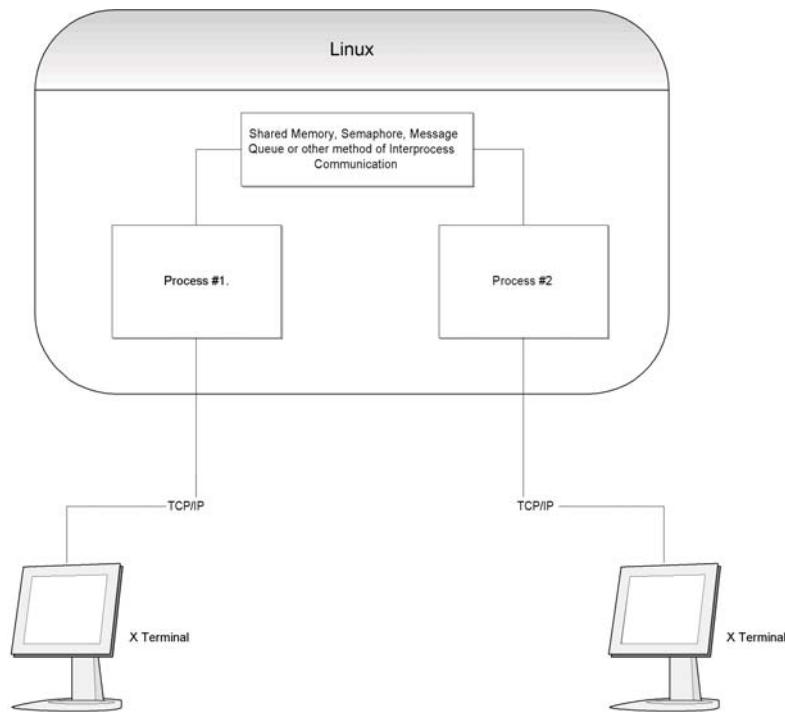
The Need to Communicate between Processes

So once you have your application up and running, you need to communicate information from one process to another. If you use the Delphi or Kylix RAD way of doing things, you may store the settings in a database or a file and allow other processes to access the information. But this may not be the best or fastest way of communicating between processes under Linux.

With Kylix, you not only have the ability to develop fast native applications under Linux, but you also have the possibility to create real-time embedded applications that may not have access to a database or with which using a database may be too much of an overhead for a real-time system.

Other possibilities may also arise in your Linux development. If you have come from a Windows background, you may still be in the mind-set that each computer runs its own applications and when you want to operate a client/server system you must have a dedicated computer system. Because Linux allows you to run a single server with many dumb terminals running from the server, you can run an application using a thin-client approach — the same way Citrix or Windows Terminal Server runs. Because the applications are all running from the same server, as shown in Figure 5.1, the communication between processes that are running on different clients across the country is actually running on the same machine.

Figure 5.1 - Interprocess communication in a thin-client environment



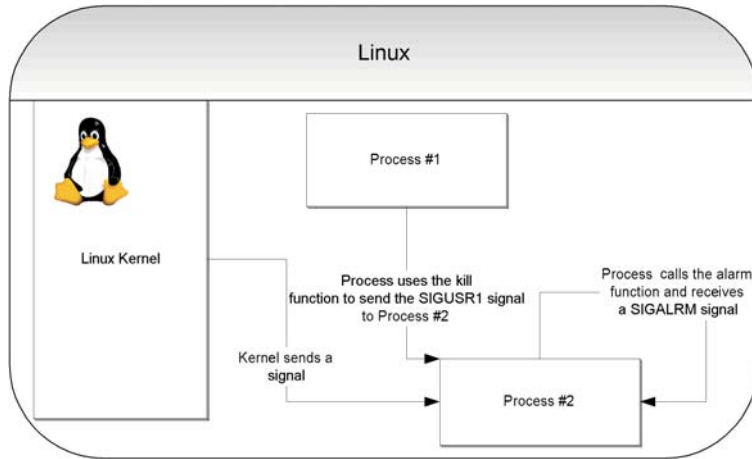
In this way, communication between different physical client machines is really a communication process that is happening on the server side. If you are running an extremely heavy load from the server, using Linux's native interprocess communication API can speed up your application.

What Methods are Available to Us?

There are several different interprocess communication methods that can be used, each having its own special place in the Linux world.

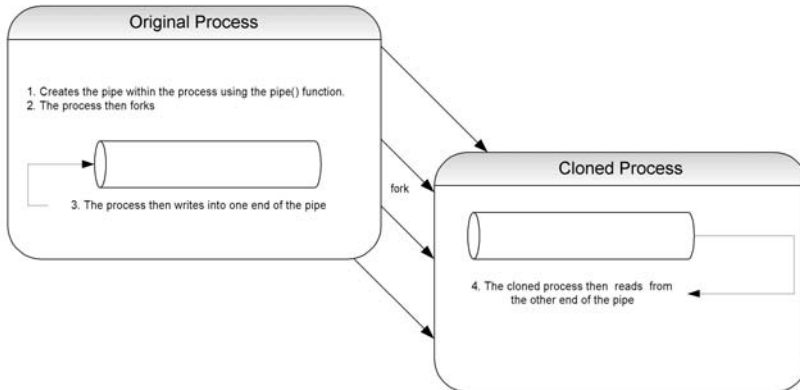
Signals, for example, are used as tokens that are passed from an application or the Linux kernel. Signals typically denote some form of error message, such as the process being aborted by a memory violation or a user hitting Ctrl+C from the console window. Normally, you would not deal with signals in most Kylix applications, especially in GUI applications. This is because much of the signal handling is taken care of for you.

Figure 5.2 -
Diagram of
signals being
passed from
one process to
another



Pipes are another method of communication between processes. If you think of a physical pipe, as shown in Figure 5.3, you know that things are placed in one end of the pipe and come out the other end of the pipe. Pipes work in the same way in the Linux API: Data goes in one end of the pipe and comes out of the other end of the pipe. In fact, pipes are created as two file descriptors where one file descriptor is used for writing and the other file descriptor is used for reading. As a communication method, pipes are normally used when a process has been forked and the file descriptors that were created are duplicated to the new child process when the original process forks.

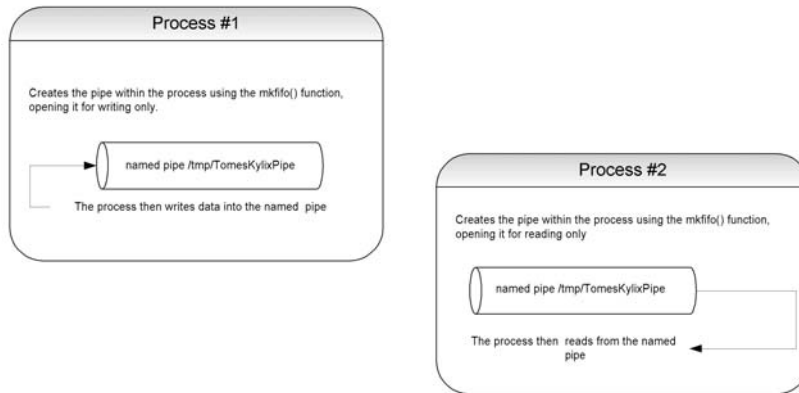
Figure 5.3 -
Pipes under
Linux



Pipes are a simple way of communicating between processes. Data goes in one file descriptor and comes out of the other file descriptor. The only problem when using pipes under Linux is that you have to fork your running process to allow a second handle to do this. This may not be practical when you are running X Windows applications such as the GUI applications that Kylix produces.

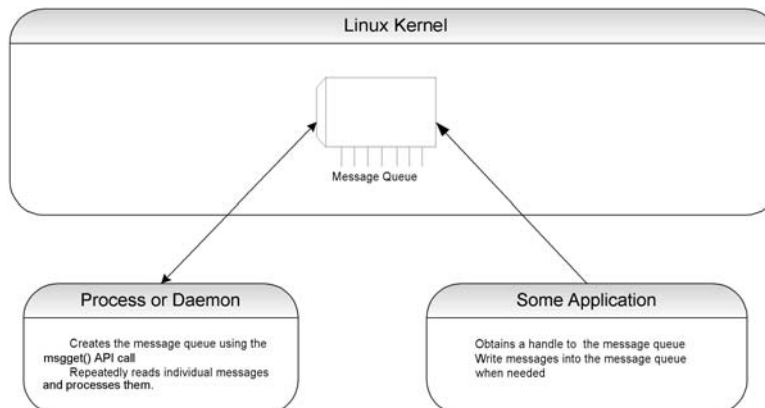
The difference between pipes and named pipes is quite obvious. Simply put, named pipes have names associated with them. This means that instead of having to create a pipe and then fork to access it, all that is needed is to know the name of the pipe itself and then you can reference it.

Figure 5.4 -
Named pipes
under Linux



Message queues are another form of interprocess communication, much like pipes or named pipes. Message queues are much easier to use than pipes because of their extremely simple API. Unlike pipes, message queues are not written to one place and read from another. Message queues allow the developer flexibility to assign priorities to messages in the queue. They also support various message retrieval methods such as retrieving the top message, receiving the message with the highest priority, receiving a message with a specific priority, or receiving a message which is not of a particular priority.

Figure 5.5 -
Linux message
queues



The last method of interprocess communication that we examine in this chapter is shared memory. This method of communication lets the Linux kernel set up a section of memory that can be accessed by many processes.

An In-depth Look at the Communication Methods

Signals — What are They?

The first communication method we will look at are signals. Signals are individual tokens that are passed either between the Linux kernel and a process or from one process to another. A signal that is sent to a process is typically sent to let the process know that an error has occurred and that the application should be aware of it. There are many signals that can be sent to a process, as shown in Listing 5.1. A signal can signify many forms of error, such as a segmentation violation (SIGSEGV), which can be caused by your application when you overflow a buffer of allocated memory, or a floating-point error (SIGFPE), which may be caused by a floating-point division by zero. A signal may even be generated by the user hitting Ctrl+C to stop the application from running.

Within Kylix, signals are handled internally. If you look in the SysUtils unit you will see that there are a group of functions that are used to control how your applications deal with exceptions while you are running a CLX application. In fact, Kylix will convert many of the signals into exceptions for you.

As a result of this, the techniques that you will see for managing signals in this chapter in most cases should only be used when writing console applications. Later, we will examine the methods to use when dealing with signals from a CLX application or shared object.

Listing 5.1 - Signal table reference

```
{ Signals. }
SIGHUP          = 1;      { Hangup (POSIX). }
SIGINT          = 2;      { Interrupt (ANSI). }
SIGQUIT        = 3;      { Quit (POSIX). }
SIGILL          = 4;      { Illegal instruction (ANSI). }
SIGTRAP        = 5;      { Trace trap (POSIX). }
SIGABRT        = 6;      { Abort (ANSI). }
SIGIOT         = 6;      { IOT trap (4.2 BSD). }
SIGBUS         = 7;      { BUS error (4.2 BSD). }
SIGFPE         = 8;      { Floating-point exception (ANSI). }
SIGKILL        = 9;      { Kill, unblockable (POSIX). }
SIGUSR1        = 10;     { User-defined signal 1 (POSIX). }
SIGSEGV        = 11;     { Segmentation violation (ANSI). }
SIGUSR2        = 12;     { User-defined signal 2 (POSIX). }
SIGPIPE        = 13;     { Broken pipe (POSIX). }
SIGALRM        = 14;     { Alarm clock (POSIX). }
SIGTERM        = 15;     { Termination (ANSI). }
SIGSTKFLT      = 16;     { Stack fault. }
SIGCHLD        = 17;     { Child status has changed (POSIX). }
SIGCLD         = SIGCHLD; { Same as SIGCHLD (System V). }
SIGCONT        = 18;     { Continue (POSIX). }
SIGSTOP        = 19;     { Stop, unblockable (POSIX). }
SIGTSTP        = 20;     { Keyboard stop (POSIX). }
SIGTTIN        = 21;     { Background read from tty (POSIX). }
SIGTTOU        = 22;     { Background write to tty (POSIX). }
SIGURG         = 23;     { Urgent condition on socket (4.2 BSD). }
SIGXCPU        = 24;     { CPU limit exceeded (4.2 BSD). }
SIGXFSZ        = 25;     { File size limit exceeded (4.2 BSD). }
SIGVTALRM      = 26;     { Virtual alarm clock (4.2 BSD). }
SIGPROF        = 27;     { Profiling alarm clock (4.2 BSD). }
```

```

SIGWINCH      = 28;      { Window size change (4.3 BSD, Sun). }
SIGIO         = 29;      { I/O now possible (4.2 BSD). }
SIGPOLL       = SIGIO;   { Pollable event occurred (System V). }
SIGPWR        = 30;      { Power failure restart (System V). }
SIGUNUSED     = 31;

```

Catching a Signal

Signals can arrive at anytime within your application. In fact, your application could be going through some huge calculation when it receives a signal. This is the very nature of signals, and you should know that you must handle signals when they arrive. But how do you deal with a signal when it arrives? You do this by setting up a procedure to receive all requests for a signal type.

The process is very similar to creating a normal Kylix event. An event in Kylix is a procedure type declared in a class. You would normally create the event by double-clicking the event handle in Kylix's Object Inspector. You could also set this event up manually as shown in Listing 5.2.

Listing 5.2 - Setting up a typical Kylix event

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Showmessage('You clicked me.');
```

```

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    //Setup the event
    Button1.OnClick := Button1Click;
end;

```

Receiving signal events is very similar. You simply need a procedure that takes a single integer parameter and uses the cdecl calling convention. You then use the signal function to specify that when a particular signal that you want to catch arrives, the procedure that you passed to the signal function will be run. This signal passed to the signal function should be one of the constants from Listing 5.1.

Listing 5.3 demonstrates using this technique to show what will happen when the SIGINT signal is sent to a process as a result of the user pressing Ctrl+C.

Listing 5.3 - Catching the SIGINT signal from a console application

```

program GettingASignal;

{$APPTYPE CONSOLE}

uses
    Libc;

procedure SignalCatcherThatGetsSIGINTSignals(signalNo: integer); cdecl;
begin
    //OK. So now we have the SIGINT Signal. Lets deal with it
    writeln('Someone hit the Control-C button....I''m melting.....');
```

```

    signal(SIGINT, TSignalHandler(SIG_DFL));
end;

begin
    //Let's write some bad code that will cause our little application
    //to have a signal thrown at it by the kernel.
    signal(SIGINT, SignalCatcherThatGetsSIGINTSignals);

    writeln('Waiting for a signal');
    pause;

    writeln('OK. We have got the signal. Wasn't that fun.');
```

end.

Figure 5.6 -
Running the
GettingASig-
nal applica-
tion

```

root@localhost.localdomain: /root/BookApplications/Programs/InterprocessCh
File Sessions Options Help
[root@localhost ConsoleApp]# ./GettingASignal
Waiting for a signal
Someone hit the Control-C button....I'm melting.....
OK. We have got the signal. Wasn't that fun.
[root@localhost ConsoleApp]#
```

The method of catching signals shown in Listing 5.3 is a little outdated and has a flaw in its implementation. The flaw is that when a process receives the signal it will automatically revert to the default method of handling the signals. As a result, there may be some time where the signals are not handled effectively. A more effective method of implementing signals is to use signal sets.

Sending a Signal to Another Process

We looked briefly at sending signals in the last chapter when we demonstrated sending the SIGKILL signal to a process to terminate the process. The unlikely function name to send a signal to a process is the kill function. Listing 5.4 demonstrates using the kill function to send a signal function to another process.

Listing 5.4 - Sending a signal to a process

```

program SendingSignals;

(*
    This application is a simple demonstration of
    sending a signal to another process.
*)
```



```

{$APPTYPE CONSOLE}

uses
  Libc;

var
  intProcessID: integer;

begin
  writeln(getpid, ': Forking the Process. ');
  intProcessID := fork;
  if intProcessID <> 0 then
    begin
      //We are the parent. Let's just hang out until we
      //get a signal
      writeln(getpid, ': Parent waiting for a signal. ');
      pause;
      writeln(getpid, ': Parent after the pause. ');
    end else begin
      //We are the child process. Let's kill the parent process
      writeln(getpid, ': Child about to send the signal. ');
      kill(getppid, SIGKILL);
      writeln(getpid, ': Child has sent the signal. ');
    end;
  end.

```

You may be wondering why the function call is called `kill`. The reason for this is that if the process does not know how to respond to the signal then the process is terminated. If a process does not deal with a signal, the default behavior is for the process to be terminated immediately. If you look at Listing 5.4, you can see where the parent process waits for a signal using the `pause` procedure. Once it has received the signal, the process and the application terminate.

It would be unwise to use signals as a method of communicating data between processes. Rather, you should use signals as a way of doing housekeeping of your application when you receive the signal notification. In this way, if a signal is received, you can release memory or resources that were previously allocated before your application terminates.

Using Signal Sets — The “Reliable” Method

The `signal` function allows you to catch signals quite effectively, but this method of using functions has its flaws. In fact, using and managing signals within your application using the `signal` function is often referred to using “unreliable signals.” The unreliability comes as a result of the signal reverting the handler to its default behavior for a signal once the signal has been received. The `signal` function does not have the ability to wait for signals and also to block signals while a signal handler is being called.

Enough about “unreliable” signal handling, let’s look at reliable signal handling.

Signal sets are known as reliable signal handling. The Linux API has the ability to create signal sets, much in the same way that you would create an integer set in Kylix. The data type that you use for working with signal sets is the `TSigSet` as shown in Listing 5.5.

Listing 5.5 - The definition of TSigSet

```

const
  _SIGSET_NWORDS = 1024 div (8 * SizeOf(LongWord));

type
  __sigset_t = record
    __val: packed array[0.._SIGSET_NWORDS-1] of LongWord;
  end;

  TSigset = __sigset_t;
  PSigset = ^TSigset;

```

You may be thinking that as you are working with sets you may assign sets of signals to an event handler. Sadly, assigning events is done the same kind of way as for the unreliable method, but instead of using the signal function you use the sigaction function.

Listing 5.6 - Using the sigaction function

```

type
  __sigaction = record
    __sigaction_handler: TSigActionHandler;
    sa_mask: __sigset_t;
    sa_flags: Integer;
    sa_restorer: TRestoreHandler;
  end;
  TSigAction = __sigaction;
  PSigAction = ^TSigAction;

function sigaction(SigNum: Integer; Action: PSigAction;
  OldAction: PSigAction): Integer; cdecl;

```

The sigaction function is used in much the same way as the signal function, in that you define what signal you want to handle and which procedure will be called when the signal is received by the process. You can see in Listing 5.6 that the sigaction function takes three parameters, the first being the signal number that it will handle. The second parameter is a TSigAction that defines what action will occur when the signal is received by the process.

The TSigAction record is used to define the handling of a particular signal. The TSigAction record holds a TSigActionHandler procedure to call when the event is retrieved, a signal set that contains the signals to block when the signal is being handled, and a list of options you can use that include the ability for the same signal to be retrieved while in a signal handler, not to receive notifications when child processes are destroyed, and to restore the default behavior of the signal once the signal has been received.

To look at a simple implementation of reliable signals, we will create a simple application that uses the alarm function that will send the process the SIGALRM signal in a specific time. The application will wait for the signal to be retrieved and then continue on. The code in Listing 5.7 demonstrates how easy it is to use signal sets.

Listing 5.7 - Using signal sets

```

program SimpleSignalSet;

{$APPTYPE CONSOLE}

```

```

uses
    Libc;

var
    MySignalSet: TSigSet;
    NewSigAction: TSigAction;
    OldSigAction: TSigAction;

procedure WeGotAnAlarm(SignalNo: integer); cdecl;
begin
    sigaction(SignalNo, @NewSigAction, nil);
    writeln('Our little process got a signal.');
```

end;

```

begin
    //Clear all the values from the signal set
    writeln('Clearing the Signal Set.');
```

sigemptyset(MySignalSet);

```

    //Let's add some signals to the signal set
    //In this demo we will listen for SIGALRM
    //which is caused by the alarm function
    writeln('Adding SIGALRM to the Signal Set.');
```

sigaddset(MySignalSet, SIGALRM);

```

    //I'll add another that we will remove later
    writeln('Adding SIGINT to the Signal Set.');
```

sigaddset(MySignalSet, SIGINT);

```

    //Let's remove the signal now from the signal set
    writeln('Removing SIGINT from the Signal Set.');
```

sigdelset(MySignalSet, SIGINT);

```

    //Let's determine if a signal is in the signal set
    if sigismember(MySignalSet, SIGPIPE) = 1 then
        writeln('SIGPIPE is a member of the set.')
```

else

```

        writeln('SIGPIPE is not a member of the set.');
```

```

    //Let's determine if a signal is in the signal set
    if sigismember(MySignalSet, SIGALRM) = 1 then
        writeln('SIGALRM is a member of the set.')
```

else

```

        writeln('SIGALRM is not a member of the set.');
```

```

    //Now let's define the action to occur when any signal in
    //the signal set is received.
    writeln('Setting up the signal.');
```

```

    //Set up the procedure to be called when the signal is received
    NewSigAction.__sigaction_handler := WeGotAnAlarm;
```

```

    //Set up a mask which defines which signals should be blocked
    //when we are processing this signal
    //For this demo we will let all signals through
    sigemptyset(NewSigAction.sa_mask);
```

```
//Set the options for the Signal
NewSigAction.sa_flags := 0;

//Now let's set up the action for the signal
writeln('Setting the sigaction');
sigaction(SIGALRM, @NewSigAction, @OldSigAction);

//What we will do is fire an alarm to go off in 5 seconds and
//let the process sleep for 10 seconds
writeln('Sending the alarm for 5 seconds.');
```

```
alarm(5);

writeln('Waiting for the signal.');
```

```
pause;
writeln('Woke up from our sleep.');
```

```
//When we get here we can now reinstate the old
//handler for the signal
writeln('Reinstating the old signal handler.');
```

```
sigaction(SIGALRM, @OldSigAction, nil);
end.
```

Just like a normal set within Kylix, there are functions to add and remove signals from a signal set, functions to determine if members are in a signal set, and many more. A listing of some of these functions with a brief description are shown in Table 5.1.

Table 5.1 - Signal set functions

Function Name	Description
sigemptyset	Clears a signal set of all values.
sigfillset	Fills a signal set with all signal values.
sigaddset	Adds an individual signal from a signal set.
sigdelset	Removes a single signal from a signal set.
sigismember	Checks to see if a particular signal is the signal set.
sigisemptyset	Checks to see if the signal has no elements.
sigandset	Combines the signals by AND'ing the values. This is the same as returning the intersection of two signal sets.
sigorset	Combines the signals by OR'ing the values. This is the same as returning the union of two signal sets.

There are many different functions that can be used to control signals within your application, which are documented in the API section of this chapter. If you are creating a CLX application, much of the signal management is performed for you.

Blocking Signals

One of the benefits of reliable signals is that you can block particular signals from being received by the process until the signal handler is finished. This is done by using the `sa_mask` field of the `TSigAction`. The process will not let any of these signals through except specific signals.

Just as you can block signals when a signal handler is executing, you can also block signals that are received by the process. These signals are called blocked signals. However, the signals that are blocked do not go away. Once the blocked signals are unblocked, your process will receive them immediately. The signals that have been blocked, sent, and are waiting to be called are called pending signals.

Blocking signals in your application is a matter of using `sigprocmask` to block and unblock signals. When your application first runs, it does not block any signals. It is up to you to determine which signals to have blocked. Using the `sigprocmask` function, you specify which operation you wish to execute. The options are to add certain signals to the set of signals that are blocked, remove signals from the set of signals that are blocked, or set exactly the signals that you want to have.

Listing 5.8 - Using the `sigprocmask` function

```
const
    SIG_BLOCK    = 0;      { Block signals.  }
    SIG_UNBLOCK  = 1;      { Unblock signals. }
    SIG_SETMASK  = 2;      { Set the set of blocked signals. }

function sigprocmask(How: Integer; SigSet: PSigset; OldSigSet: PSigset): Integer;
    cdecl;
```

Using the `SIG_BLOCK` option will add the signals in `SigSet` to the list of the signals that are being blocked. Using `SIG_UNBLOCK` will remove the entries in `SigSet` from the set of signals that are currently being processed, even if there are signals in `SigSet` that are not in the set of signals being blocked. When the `SIG_SETMASK` option is used, the set of signals is set to the set contained in `SigSet`. When the `OldSigSet` value is not nil, the set of signals that were blocked before the function was called are copied into `OldSigSet`.

So putting the `sigprocmask` function to use is a matter of simply defining the set of signals and then blocking them as shown in Listing 5.9.

Listing 5.9 - Blocking signals using the `sigprocmask` function

```
procedure TfrmBlockingSignalsTestForm.btnBlockTheSignalsClick(Sender: TObject);
var
    SignalsToBlock: TSigset;
    OldSignals: TSigset;
begin
    //This function will block the user-defined signals.

    sigemptyset(SignalsToBlock);

    //Add the custom signals to the signal set
    sigaddset(SignalsToBlock, SIGUSR1);
    sigaddset(SignalsToBlock, SIGUSR2);

    //Block the current messages.
    sigprocmask(SIG_BLOCK, @SignalsToBlock, @OldSignals);
end;
```

As mentioned before, once you block a signal it does not go away. Once a signal that has been sent to a process is unblocked, the signal will be processed almost immediately. It

may be that you will need to examine the blocked signals that are waiting to be processed. This is done using the `sigpending` function. If you are blocking a signal and you are about to unblock the signal, you may need to determine if you want to handle a particular handler for the signal before it is unblocked, so that your process will not terminate if you do not handle the signal.

Listing 5.10 - Using the `sigpending` function

```
function sigpending(var SigSet: TSigset): Integer; cdecl;

//Using the sigpending function.
procedure TfrmBlockingSignalsTestForm.btnViewPendingSignalsClick(Sender: TObject);
var
    pendingSignals: TSigset;
    counter: integer;
begin
    sigpending(pendingSignals);

    //Display the details about the signals that are pending.
    mmPendingSignals.Lines.Clear;
    for counter := 0 to _NSIG - 1 do
        if sigismember(pendingSignals, counter) = 1 then
            mmPendingSignals.Lines.Add('Pending Signal - '+
                StrPas(strsignal(counter)));
    end;
```



Note: The signal set that is blocked is called the signal mask and is a valid `TSigSet`. In this chapter we only deal with the functions that are known to be reliable. Blocking signals can be done using the `sigblock` and `sigsetmask` functions; however, these functions use a signal mask with an integer value and these functions are not guaranteed to contain all the possible signals. As a result, it is much better if your application blocks signals using the functions that make use of a `TSigSet` data type.

Using Signals within Kylix Applications

Within Kylix internally, signals are handled using reliable signals. If you are creating an application and it uses the `SysUtils` unit, then by default Kylix will convert particular signals into exceptions. For example, `SIGINT`, which is retrieved by a user halting the process by pressing `Ctrl+C`, will be converted to an `EControlC` exception.

But this handling is only done on applications within Kylix. If you are creating a shared object, such as one that is used in an Apache module or a plug-in to another system, then the conversions of signals is not done for you. What needs to happen is that your shared object needs to connect the signal-to-exception handling routines into the application. This is done by calling `HookSignal(RTL_SIGDEFAULT)` in the unit's initialization section to set Kylix's run-time library to handle the errors and then `UnhookSignal(RTL_SIGNAL-DEFAULT)` in the unit's finalization section as shown in Listing 5.11.

Listing 5.11 - Hooking signals to RTL exceptions in a shared object

```

unit Unit1;

interface

uses
  Variants, SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  WebModule1: TWebModule1;

implementation

uses WebReq;

{$R *.xfm}

initialization
  WebRequestHandler.WebModuleClass := TWebModule1;
  HookSignal(RTL_SIGDEFAULT);
finalization
  UnhookSignal(RTL_SIGNALDEFAULT);
end.

```

Also, signals may have some problems if you are running a system that causes some other issues, for example, if you have a shared object, such as the Apache DSO object, that calls another shared object that may have been written in C. Both of these shared objects may install their own signal handlers. To avoid this scenario, make sure that shared objects are unloaded in the reverse order that they were loaded.

For the above problem, there are also a group of functions in the SysUtils unit that allows you to test whether Kylix is actually managing the signal handling in your shared object. The functions are displayed below.

Listing 5.12 - Using the Kylix SysUtils signal functions

```

type
  TSignalState = (ssNotHooked, ssHooked, ssOverridden);

function InquireSignal(RtlSigNum: Integer): TSignalState;
procedure HookSignal(RtlSigNum: Integer);
procedure UnhookSignal(RtlSigNum: Integer; OnlyIfHooked: Boolean = True);
procedure AbandonSignalHandler(RtlSigNum: Integer);

```

The first of these functions, `InquireSignal`, returns if a signal is currently being managed by the Kylix RTL, returning `ssNotHooked` if the Kylix RTL is not managing it, `ssHooked` if

Kylix RTL is managing it, and `ssOverridden` if another shared object has taken control of the signal.

The `HookSignal` function will make Kylix's RTL convert the signal into an exception, where `UnhookSignal` will revert a previously hooked signal back to its original state and the `AbandonSignalHandler` function removes a signal's ability to be reinstated to its previous handler.

Programming Pipes

Pipes are another fantastic form of interprocess communication. Pipes work on the principal of a physical pipe. That is, stuff goes in one end of the pipe and the same stuff comes out of the other end. When it comes to pipes under Linux, the stuff that comes in and goes out of the pipes is data and each end of the pipe is represented by a file descriptor.

Where the data goes in is represented by a write-only file descriptor. The reverse is also true: Where data comes out of the pipe is represented by a read-only file descriptor. It is quite a simple process to create a pipe within your application. All that one needs to do is call the pipe function, which takes either a single `TPipeDescriptors` variable or a pointer to an array of at least two integer elements. These integer elements represent the file descriptor that will be passed back into the integer array elements. See Listing 5.13 to see the versions of the pipe function that are available to you.

Listing 5.13 - Using the pipe functions

```
type
  TPipeDescriptors = {packed} record
    ReadDes: Integer;
    WriteDes: Integer;
  end;

function pipe(var PipeDes: TPipeDescriptors): Integer; cdecl; overload;
function pipe(PipeDes: PInteger): Integer; cdecl; overload;
```

You can see in Listing 5.13 that regardless of which pipe function you call, you still get back two file descriptors — one for reading and the other for writing, in that order. Passing in the `TPipeDescriptors` variable would be normally preferred within your application as it makes the application more readable. You are more likely to understand a write operation to `MyPipe.ReadDes` than to some integer that represents a file descriptor.

Creating a pipe in itself is not all that is required to use it for interprocess communication. The procedure for setting up processes for interprocess communication using pipes is shown in Figure 5.8.

The process is quite simple. First, your process will create the pipe that it will use for data transfer. After that is done, your process should fork. This will give both copies of the process access to the pipes due to the duplication that occurs during forking. Once both processes have access to the file descriptors, you can set one process to write data into the pipe and have the other process read the data as shown in Listing 5.14. Figure 5.3 also demonstrates the steps involved in creating the pipe communication process.

Listing 5.14 - A simple program that makes use of pipes

```

program SimplePipeDemo;

{$APPTYPE CONSOLE}

(*
    This application demonstrated the use of pipes within
    an application. We can only really use pipes within
    a console application due to its reliance on forking
    to allow both processes to communicate.
*)

uses
    Libc;

type
    TCustomer = record
        Name: array[0..30] of char;
        Age: integer;
        Email: array[0..45] of char;
    end;

var
    intChildProcess: integer;
    MyPipe: TPipeDescriptors;
    SomeCustomer: TCustomer;
    intDataLength: integer;

begin
    //First we'll create the pipe
    writeln(getpid, ': Creating the Pipes');
    if pipe(MyPipe) <> 0 then
        begin
            //The pipe creation was not successful.
            writeln(getpid, ': The pipes could not be created');
            Exit;
        end;

    //Now we fork the process so that both processes
    //can have access to the pipe.
    writeln(getpid, ': Forking the process');
    intChildProcess := fork;

    if intChildProcess = -1 then
        begin
            //We have an error with the fork issues. Let's exit the app
            writeln(getpid, ': There was a problem forking the process');

            //Close the pipes and stop
            _close(MyPipe.ReadDes);
            _close(MyPipe.WriteDes);

            Exit;
        end;
end;

```

```

if intChildProcess = 0 then
begin
    //We are the parent process. Let's write some stuff to the pipe.
    writeln(getpid, ': The parent is now writing to the pipe');

    //Because we are writing to the pipe this process doesn't need the
    //read pipe.
    __close(MyPipe.ReadDes);

    //Let's write a few customers to the pipe
    with SomeCustomer do
    begin
        Name := 'John Doe';
        Age := 25;
        Email := 'johndoe@notreal.com';
    end;
    __write(MyPipe.WriteDes, SomeCustomer, sizeof(TCustomer));

    with SomeCustomer do
    begin
        Name := 'Jane Doe';
        Age := 23;
        Email := 'janedoe@notreal.com';
    end;
    __write(MyPipe.WriteDes, SomeCustomer, sizeof(TCustomer));

    //Now let's close the writing pipe for this process
    __close(MyPipe.WriteDes);
end else begin
    //We are the child process. Let's read from the pipe.
    writeln(getpid, ': The child process is now reading from the pipe');

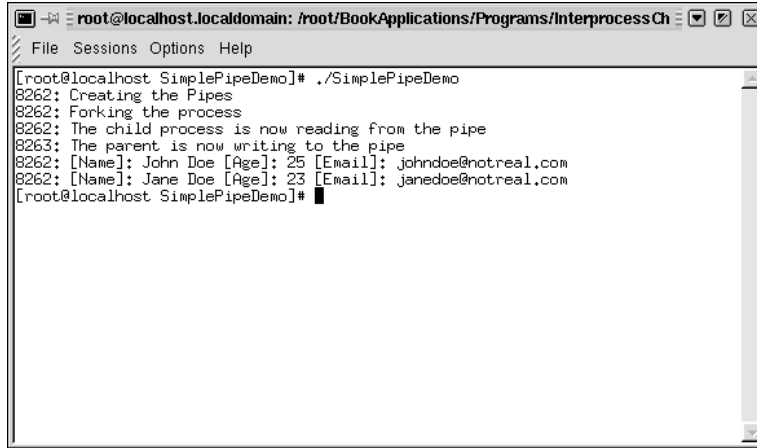
    //This process doesn't need the write pipe, so we'll close it
    __close(MyPipe.WriteDes);

    //Let's read the entries from the Pipe, one record at a time.
    intDataLength := __read(MyPipe.ReadDes, SomeCustomer, sizeof(TCustomer));
    while intDataLength <> 0 do
    begin
        //Let's display the information about the customer
        writeln(getpid, ': [Name]: ', SomeCustomer.Name, ' [Age]: ', SomeCustomer.Age,
            ' [Email]: ', SomeCustomer.Email);
        intDataLength := __read(MyPipe.ReadDes, SomeCustomer, sizeof(TCustomer));
    end;

    //Now let's close the reading pipe for this process
    __close(MyPipe.ReadDes);
end;
end.

```

Figure 5.7 -
The output
of the
SimplePipe-
Demo
application



```

root@localhost.localdomain: /root/BookApplications/Programs/InterprocessCh
File Sessions Options Help
[root@localhost SimplePipeDemo]# ./SimplePipeDemo
8262: Creating the Pipes
8262: Forking the process
8262: The child process is now reading from the pipe
8263: The parent is now writing to the pipe
8262: [Name]: John Doe [Age]: 25 [Email]: johndoe@notreal.com
8262: [Name]: Jane Doe [Age]: 23 [Email]: janedoe@notreal.com
[root@localhost SimplePipeDemo]#

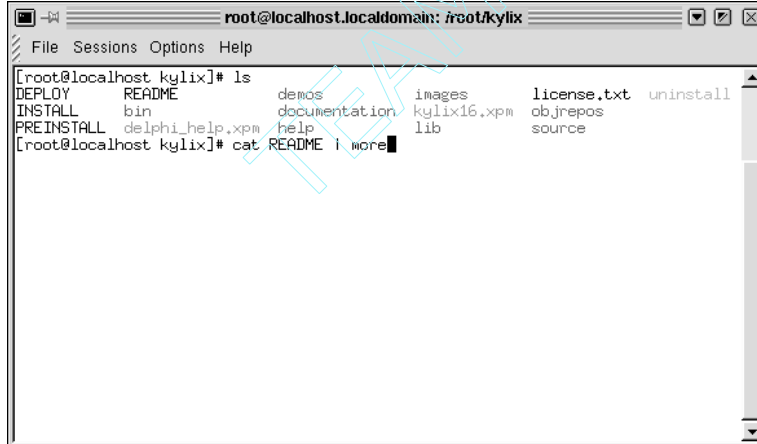
```

Listing 5.14 demonstrates a simple pipe application, but you may have different requirements of the communicating processes.

You may, for example, need to set up two pipes that can be used from within your application. In this way, one pipe can be used as a method of talking between the server and the client and the other pipe can be used to talk from the client to the server.

This is just a small taste of what you can do with pipes within your application.

Figure 5.8 -
Using pipes
within the
Linux OS



```

root@localhost.localdomain: /root/kylix
File Sessions Options Help
[root@localhost kylix]# ls
DEPLOY      README      demos      images      license.txt  uninstall
INSTALL     bin         documentation  kylix16.xpm  objrepos
PREINSTALL  delphi_help.xpm  help      lib         source
[root@localhost kylix]# cat README | more

```

You may have already heard of the terminology of pipes before. In fact, Linux itself makes use of the functionality of pipes to redirect standard input as demonstrated in Figure 5.8. The pipe symbol (`|`) in the figure tells the operating system to use the output of the `cat README` operation as the standard input for the `more` command.

This is a classic demonstration of the use of pipes where data is written in one end and read from another. This classic use of redirection is great when used with the Linux shell, but it can also be achieved within your applications using the `popen` function.

```
function popen(const Command: PChar; Modes: PChar): PIOFile; cdecl;
```

The `popen` function is an extremely handy way of creating a pipe without going through the entire forking process. The `popen` function executes the Linux command specified in the Command parameter.

Which mode you open the pipe with specifies the kind of file stream that the function will return. You have the option of opening the file for reading or writing, but as it is a pipe, you do not have the option of opening it for both reading and writing. If the Modes parameter specified reading using the `r` value, the stream that the function returns is actually the standard output from the executed command. Likewise, if the parameter to the Modes parameter is to write to the file using `w`, the stream that is returned by the function will be used as the standard input for the executed shell.

As the `popen` function returns a stream handle, you may think that to close the returned stream you would use the `fclose` API call. This is not this case. It is important that the stream be closed with the `pclose` function rather than the `fclose` function.

Due to the fact that making use of pipes will require you to fork your process, pipes are not a good choice when you want to use them as a method of communication between processes that are GUI based. Just using `fork`, `popen`, or any of the `exec` functions will result in your application forking. Because of this, it is best to restrict the use of pipes to console applications due to their reliance on duplicating pipe file descriptors to new processes. Rather than rely on file descriptors, named pipes allow you to create a pipe but not have the reliance on file descriptors. All that is needed is the name of the pipe that is being made.

Named Pipes — A Better Alternative

Named pipes are like normal Linux pipes with one major difference. Normal pipes are created by a pipe in one process, forked, and then used in both processes. With named pipes, you no longer have to fork a process in order to use pipes. The named pipe itself is created using an agreed upon name by both processes, hence the term “named pipes.” Using named pipes, you don’t go through the process of forking a process and then using the duplicated file descriptors.

Named pipes are also referred to as FIFOs, which stands for file-in, file-out. This is due to the fact that when a named pipe is created, a file on the Linux system is actually created. As a result, when specifying the name of the named pipe, you will have to make sure that the processes that want access to the pipe’s information have access to the pipe. This is achieved either by having both the client and the server applications running from the same directory and referencing a named pipe, such as `MyPipe`, or by specifying the full path for the named pipe, which could be something like `/opt/myproduct/mypipe`.

To create a named pipe, the named pipe must first be created. This is done using the `mkfifo` function. The `mkfifo` function takes the location of the named pipe that is about to be created and the permissions for the named pipe.

There are several steps that need to be done to be sure that a named pipe can be used. First of all, you need to make sure that the named pipe has been created before you go any further using the `mkfifo` function.

```
function mkfifo(PathName: PChar; Mode: __mode_t): Integer; cdecl;
```

The parameters that are passed to the `mkfifo` function are the name of the pipe in the `PathName` parameter and the permission mask that specifies who can access the named pipe in the `Mode` parameter. The `Mode` parameter is the same permission mask that is used when opening a file descriptor with the values listed in Table 3.2. What is returned from the `mkfifo` function is a file descriptor that you can use with normal `__read` and `__write` Linux API calls.

As the named pipe is really just a special file on the Linux file system, the `mkfifo` function creates the named pipe. To use the named pipe you need to open the pipe by using the Linux API function `open`. The parameters to the `open` function depend on what end of the pipe you have. If you call `open` for reading only, you will have the reading end of the pipe. Likewise, if you call `open` for writing, you will have access to the end of the pipe that writes the data into the pipe.

Listings 5.15, 5.16, and 5.17 demonstrate a simple process that makes use of pipes so that they can communicate. In this application the reader application from Listing 5.16 is responsible for creating the named pipe, opening the named pipe for reading, and reading data from the fifo at specific intervals.

So when the writer application starts and writes data into the named pipes, the reader application then has the ability to read data from the named pipes that was written by the writer application in Listing 5.17. As a result of the interaction, we have a simple form of interprocess communication with named pipes. As you can see from the examples, the use of named pipes will work well with CLX applications, making it a good choice for any simple interprocess communication process.

Listing 5.15 - LogFIFOTypes

```
unit LogFIFOTypes;

interface

type
    TLogEventNotification = record
        LogEventName: string[50];
        LogTime: TDateTime;
        TerminalID: integer;
    end;

const
    PIPENAME = 'TomesKylixPipe';

implementation

end.
```

Listing 5.16 - unitMainReaderForm

```
unit unitMainReaderForm;

interface

uses
```

SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
QStdCtrls, QExtCtrls, QComCtrls, QTypes;

```

type
  TfrmReader = class(TForm)
    Panel1: TPanel;
    Label1: TLabel;
    tmrPipe: TTimer;
    sbStatus: TStatusBar;
    mmLogEntries: TMemo;
    procedure tmrPipeTimer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    NamedPipeHandle: integer;
    MessagesRecieved: integer;
    procedure IncreaseMessageCount;
  end;

var
  frmReader: TfrmReader;

implementation

{$R *.xfm}

uses
  Libc, LogFIFOTypes;

procedure TfrmReader.FormCreate(Sender: TObject);
begin
  //Here we will create our Named Pipe
  if mkfifo(PIPENAME, 666) = -1 then
    begin
      //We could not create the named pipe. It may be that the
      //pipe already exists.
      if errno <> EEXIST then
        begin
          //The FIFO could not be created by a reason other
          //than it already exists, so we'll exit the
          //application gracefully.
          Application.Terminate;
        end;
      end;
    end;

  //Now that we have got this far, we can start reading
  //from the named pipe. We need to open the file for reading though
  NamedPipeHandle := open(PIPENAME, O_RDONLY or O_NONBLOCK);
  if NamedPipeHandle = -1 then
    begin
      //We could not open the FIFO
      Application.Terminate;
    end;

  MessagesRecieved := 0;
  tmrPipe.Enabled := true;
end;

```

```

procedure TfrmReader.IncreaseMessageCount;
begin
    inc(MessagesRecieved);
    sbStatus.SimpleText := 'Read '+inttostr(MessagesRecieved)+' messages';
end;

procedure TfrmReader.tmrPipeTimer(Sender: TObject);
var
    msg: TLogEventNotification;
    intLength: integer;
begin
    //Every time the timer starts, we will attempt to
    //read one of our customer records from the named pipe.
    intLength := __read(NamedpipeHandle, msg, sizeof(TLogEventNotification));

    //We'll just read one message at a time from the FIFO
    if intLength = sizeof(TLogEventNotification) then
    begin
        //Display the details of the current message
        mmLogEntries.Lines.Add(Format('[Terminal %d]: %s - %s',
            [msg.TerminalID, FormatDateTime('ddd dd/mm/yyyy hh:nn:ss', msg.LogTime),
            msg.LogEventName]));

        IncreaseMessageCount;
    end;
end;

end.

```

Listing 5.17 - unitMainWriterForm

```

unit unitMainWriterForm;

interface

uses
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
    QStdCtrls, Libc, LogFIFOTypes;

type
    TfrmLogFIFOWriter = class(TForm)
        Label1: TLabel;
        Label2: TLabel;
        Label3: TLabel;
        lblTerminalID: TLabel;
        lblDateSent: TLabel;
        edEventName: TEdit;
        btnSend: TButton;
        procedure btnSendClick(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure FormDestroy(Sender: TObject);
    private
        TerminalID: integer;
    public
        intFIFOHandle: integer;

```

```

        SomeEvent: TLogEventNotification;
    end;

var
    frmLogFIFOWriter: TfrmLogFIFOWriter;

implementation

{$R *.xfm}

procedure TfrmLogFIFOWriter.btnSendClick(Sender: TObject);
begin
    //Fill up some test data
    SomeEvent.LogEventName := edEventName.Text;
    SomeEvent.LogTime := Now;
    SomeEvent.TerminalID := TerminalID;

    //Lets send the message to the FIFO
    __write(intFIFOHandle, SomeEvent, sizeof(TLogEventNotification));

    //Close the FIFO
    __close(intFIFOHandle);
end;

procedure TfrmLogFIFOWriter.FormCreate(Sender: TObject);
begin
    //For the Terminal ID, what we will do is simply
    //use the process ID.
    TerminalID := getpid;
    lblTerminalID.Caption := inttostr(TerminalID);

    //Here we will open the FIFO.
    intFIFOHandle := open(PIPENAME, O_WRONLY);
    if intFIFOHandle = -1 then
    begin
        //We have a problem with opening the FIFO
        MessageDlg('There was a problem opening the FIFO',
            mtWarning, [mbOk], 0);
        Application.Terminate;
    end;
end;

procedure TfrmLogFIFOWriter.FormDestroy(Sender: TObject);
begin
    __close(intFIFOHandle);
end;

end.

```

Message Queues — A Better Way

The different methods of interprocess communication with Linux all have strengths and weaknesses. Signals are messages, and are not effective for sending chunks of data between processes. In addition, signals are not very good as a means of communication between

CLX applications. Pipes and named pipes only allow you reading and writing from one place.

Message queues are another form of interprocess communication that set up a queue so that data can be written and read from any process without having to go through any forking processes. Message queues have many advantages over other forms of interprocess communication such as a simplified set of API functions and the ability to alter the level of security of the queue while it is still in use. One of the best advantages of message queues is that they are the only method of interprocess communication that is persistent. This is a major advantage over the other methods as you may have a scenario in which there are a large number of processes accessing the message queue.

Message queues also have the advantage that every message that is placed in the message queue has a priority associated with it. As a result, you have the option of retrieving messages of a specific priority rather than reading the next message from the queue. In fact, there are several options available, including the ability to read the first message, read the message of the highest priority, read a message of a particular priority, or read the next message regardless of priority.

There are four functions that are used to operate message queues within your applications, as shown in Listing 5.18.

Listing 5.18 - Using the four message queue API functions

```
function msgget(__key: key_t; __msgflg: Integer): Integer; cdecl;
function msgsnd(__msqid: Integer; const Msgp; __msgsz: size_t;
  __msgflg: Integer): Integer; cdecl;
function msgrcv(__msqid: Integer; var Msg; __msgsz: size_t;
  __msgtyp: Longint; __msgflg: Integer): Integer; cdecl;
function msgctl(__msqid: Integer; __cmd: Integer; __buf: PMsgQueueIdDesc): Integer;
  cdecl;
```

The four functions are relatively simple to use and give you complete control of the message queue. For a complete reference to these functions, examine the API reference later in this chapter.

The first function listed in Listing 5.18 is the msgget function. This function will either create a message queue or connect to an existing message queue. It will return a message queue ID, which is used to call the other message queue API functions.

Messages are sent and received from the message queue using the msgsnd and msgrcv API function calls. The messages that are placed in the message queue are chunks of data that are normally represented by typical Pascal records. Every message that is placed into the queue needs to begin with an integer value to represent the priority of the message. So if we were sending and receiving customer records you would use the record structure shown in Listing 5.19.

Listing 5.19 - An example record for use in a message queue

```
Type
  TCustomerRecord = record
    Priority: longint;
    CustName: string[40];
    CustEmail: string[60];
    Age: integer;
  end;
```

Now a structure such as the one in Listing 5.19 would be passed into the `msgsnd` API call to send a message. The `msgsnd` function also requires the size of the chunk of data that is being sent, but it does not consider the priority to be part of the message size. So, to calculate the size of the record that you pass into the function you would use the following:

```
mqSizeOfMessage := sizeof(TCustomerRecord) - sizeof(longint);
```

The same rules about the record size apply when receiving messages from the message queue using the `msgrcv` function. The `msgrcv` function will retrieve a message from the message queue if there is one. The `msgrcv` function also allows you to specify what type of message you want to receive from the message queue. If no message is in the queue, the `msgrcv` function will wait until a message is in the queue before returning from the function call. This allows you to create a message queue server that will have a minimal impact on your Linux machine.

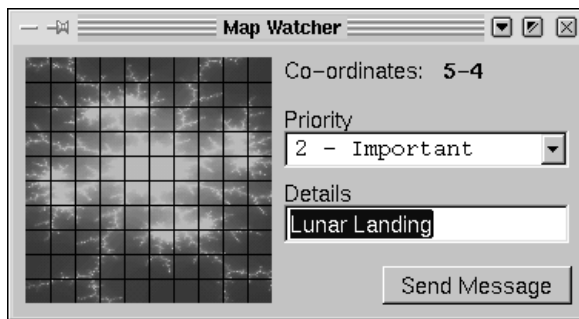
The last of the functions we will look at is the `msgctl` function. This function allows you to do a variety of things, such as obtain information about the queue, reassign ownership of the queue, and remove a message queue.

One great advantage of message queues is their persistence. Once a queue is created, it is stored within the kernel. As a result, you do not need to have both the client and server running at the same time. If you only have the client running, the messages will be sent to the queue to wait for a process to begin reading the messages from the queue.

Putting all of this together into an application involves several steps, the first being that you should define the messages that you will be sending and receiving from the message queue.

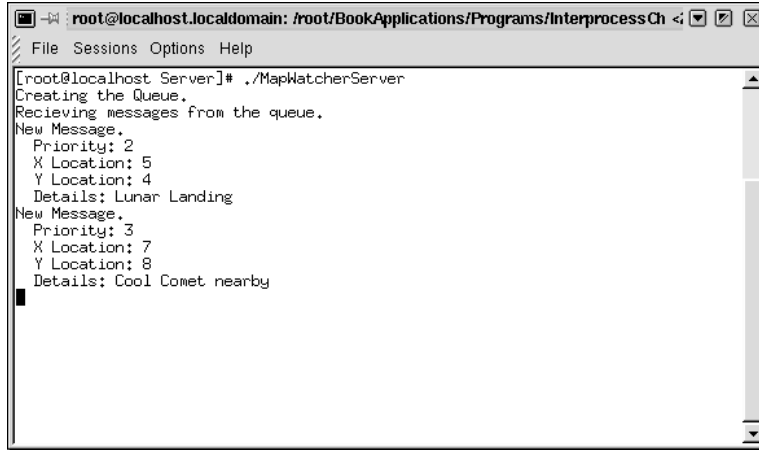
In our example, we will present a simple communication system that looks at a map and sends coordinates of where any security alerts are raised. The first application that will be created will set up the message queue and look for messages on the queue and will look for messages of the highest priority. Our other application will write messages to the message queue as demonstrated in Figure 5.9.

Figure 5.9 -
The message
queue client
application



Listing 5.21 demonstrates writing a server application that is used to read from a message queue. The demonstration is quite simple. You have a console application that reads the most important messages one at a time and outputs the details of the message received. A typical server such as this would be well suited to be run as a daemon process using the techniques in Chapter 4.

Figure 5.10 -
The server
application



Listing 5.20 - Common unit shared by both the message queue client and server applications

```
unit MapWatcherTypes;

interface

type
    TMapWatcherEntry = record
        Priority: longint;
        XLocation: integer;
        YLocation: integer;
        Details: string[40];
    end;

const
    QUEUEIPCVALUE: integer = 2002;

implementation

end.
```

Listing 5.21 - Using the message queue server application

```
program MapWatcherServer;

{$APPTYPE CONSOLE}

(*
    This application will just sit and listen
    for messages in the message queue. When it retrieves
    a message it will just output the message to the screen.
    You could do so much more than this, but this app
    simply demonstrates the use of message queues.

    An application like this would be perfect to be run as
    a daemon process that sits quietly in the background.
*)

uses
```

```

Libc,
LibHelperFunctions,
MapWatcherTypes in '../Common/MapWatcherTypes.pas';

var
  queueHandle: integer;
  MapWatcher: TMapWatcherEntry;
  GotMessage: boolean;

begin
  //Let's Open the message queue, and create it if needed
  writeln('Creating the Queue.');
```

queueHandle := msgget(QueueIPCVALUE,
 CreatePermissionsMask(upAllPermissions) or IPC_CREAT);
 if queueHandle = -1 then
 begin
 //The queue could not be created
 perror('The Message Queue could not be created.');

Exit;
 end;

writeln('Receiving messages from the queue.');

//Now let's try and get all the messages that we can.
 repeat
 //The msgrcv function will wait until the next message is sent to the queue.
 GotMessage := msgrcv(queueHandle, MapWatcher,
 sizeof(TMapWatcherEntry) - sizeof(longint), -10, 0) <> -1;

if GotMessage then
 begin
 //Let's display the information
 writeln('New Message.');

writeln(' Priority: ', MapWatcher.Priority);
 writeln(' X Location: ', MapWatcher.XLocation);
 writeln(' Y Location: ', MapWatcher.YLocation);
 writeln(' Details: ', MapWatcher.Details);
 end;

until not GotMessage;

//Let's clean up the message Queue
 writeln('Deleting the queue.');

msgctl(queueHandle, IPC_RMID, nil);
end.

Listing 5.22 - Using the message queue client application

```

unit unitMapWatcherForm;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls, QExtCtrls, Libc, LibHelperFunctions;

type
  TfrmMain = class(TForm)
    pbMap: TPaintBox;
```

```

    Label1: TLabel;
    cbPriority: TComboBox;
    Label2: TLabel;
    edDetails: TEdit;
    btnSend: TButton;
    Label3: TLabel;
    lblCoordinates: TLabel;
    procedure pbMapPaint(Sender: TObject);
    procedure pbMapMouseDown(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
    procedure btnSendClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
private
    XPos, YPos: integer;
public
    bm: TBitmap;
end;

var
    frmMain: TfrmMain;

implementation

uses MapWatcherTypes;

{$R *.xfm}

procedure TfrmMain.pbMapPaint(Sender: TObject);
var
    x, y: integer;
begin
    //Draw the map on the Paint Box
    pbMap.Canvas.Rectangle(pbMap.ClientRect);

    if Assigned(bm) then
        pbMap.Canvas.Draw(0, 0, bm);

    for x := 1 to 10 do
    begin
        pbMap.Canvas.MoveTo(x * 16, 0);
        pbMap.Canvas.LineTo(x * 16, 160);
    end;

    for y := 1 to 10 do
    begin
        pbMap.Canvas.MoveTo(0, y * 16);
        pbMap.Canvas.LineTo(160, y * 16);
    end;

end;

procedure TfrmMain.pbMapMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    //Set the co-ordinates

```

```

XPos := X div 16 + 1;
YPos := Y div 16 + 1;

lblCoordinates.Caption := Format('%d-%d', [XPos, YPos]);
end;

procedure TfrmMain.btnSendClick(Sender: TObject);
var
  queueHandle: integer;
  Msg: TMapWatcherEntry;
begin
  if cbPriority.ItemIndex <= 0 then
  begin
    Showmessage('Please select a priority before continuing.');
```

Exit;

```

  end;

  //Send the message to the message queue.
  queueHandle := msgget(QueueIPCValue,
    CreatePermissionsMask(upAllPermissions) or IPC_CREAT);

  if queueHandle = -1 then
  begin
    ShowMessage('oops. Sorry but I couldn't create a link to the message queue.');
```

Exit;

```

  end;

  //Now let's send the message
  with Msg do
  begin
    Priority := cbPriority.ItemIndex;
    XLocation := XPos;
    YLocation := YPos;
    Details := edDetails.Text;
  end;

  if msgsnd(queueHandle, Msg, sizeof(TMapWatcherEntry) - sizeof(longint), 0) = -1 then
  begin
    Showmessage('The message could not be sent to the message queue.');
```

end;

```

end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  try
    bm := TBitmap.Create;
    bm.LoadFromFile('mapimage.bmp');
```

except

```

    Showmessage('The map image could not be loaded.');
```

end;

```

end;

end.
```

Semaphores — Communicating Operation Status Between Processes

Semaphores are a communication method unlike pipes and message queues. When used for interprocess communication, a semaphore is used as a token to say that a particular operation is being accessed by a process or thread and it requires exclusive access to the resource.

To draw an analogy, consider the case of two chefs in a kitchen where there is only one oven. Both of these chefs need access to the oven to make their meals. But for practical purposes only one chef can have access to the oven at a time. So, the chefs devise a system that when a plate is placed on the top of the oven the oven is in use and cannot be used by anyone until the plate is removed.

In the real world, accessing a single resource from multiple processes is very similar to the chef scenario. The chefs represent processes who have work to do and both want to access the same resource. The plate on top of the stove corresponds to a semaphore.

As you can see, a semaphore is really just a mechanism for displaying the state of a particular resource. The example demonstrates a semaphore that is protecting only a single resource, but what if there were three places in the oven where meals could be cooked? In the chefs' world, a simple solution could be that for every place in the oven that is in use a plate is placed on the oven. That way if there is one plate on the oven, the other chefs know that there are two more available places in the oven.

Semaphores have a built-in counter system similar to this mechanism. A semaphore initializes itself with a count of how many resources are available. So, if a semaphore is protecting five resources and initializes itself to one, only one of the five resources is available. The operations that you will perform on a semaphore are decreasing the semaphore's value, which represents the taking of a resource, or adding to the counter, which represents a resource becoming available.

Programming Semaphores

Listing 5.23 - Using the Linux semaphore functions

```
function semget(__key: key_t; __nsems: Integer; __semflg: Integer): Integer; cdecl;
function semop(__semid: Integer; __sops: PSemaphoreBuffer; __nsops: size_t): Integer;
    cdecl;
function semctl(__semid: Integer; __semnum: Integer; __cmd: Integer): Integer; cdecl;
    varargs;
```

If you thought that the API functions for working with the message queue API was easy, you're going to enjoy working with semaphores. Semaphores have a small collection of functions, which are shown in Listing 5.23, that give you the ability to create, open, and perform operations.

The first of these functions is the `semget` function. Its purpose in life is to either create new semaphores or open a reference to existing semaphores. You pass into the semaphore the number of resources it will be protecting as well as the permission flags specifying which users will have access to the semaphore. In most cases, you will only want to create a single semaphore, but you also have the ability to create an array of semaphores. Obtaining the semaphore handle is as easy as the code in Listing 5.24.

Listing 5.24 - Obtaining a semaphore handle

```
SemaphoreHandle := semget(strtoint(edSemaphoreIPCID.Text),
    strtoint(edAmountOfResources.Text),
    CreatePermissionsMask(upEveryoneCanExecute) or IPC_CREAT);
```

After the semaphore handle is obtained, the application needs to initialize the value of the semaphore counter. Once a semaphore has been created, the value of the semaphore counter is set to zero, which means that all the resources that the semaphore is protecting have been allocated. Normally when your application initializes, this is not the case, so you need to set the value of the semaphore counter. This is done with the `semctl` function which also performs many control operations, such as obtaining information about the values of the semaphore, getting and setting the value of the semaphore, and removing the semaphore. Listing 5.25 demonstrates initializing a semaphore's value, and Listing 5.26 demonstrates obtaining the value of a semaphore.

Listing 5.25 - Initializing a semaphore

```
if semctl(SemaphoreHandle, 0, SETVAL, strtoint(edAmountOfResources.Text)) = -1 then
begin
    Showmessage('The semaphore could not be initialised.');
```

```
    Exit;
end;
```

Listing 5.26 - Obtaining the counter of a semaphore

```
ResourcesAvailable := semctl(SemaphoreHandle, 0, GETVAL);
```

The last of the API functions that we will look at is the `semop` function, which performs either a single operation or a list of operations on a semaphore. The only useful operation that can be used with the `semop` function is to change the value of a given semaphore, either by increasing the value to specify that the semaphore has a new resource to manage or by decreasing the value of the semaphore to signify that a resource has been taken. To perform this operation, you will need to pass a `TSemaphoreBuffer` record into the `semop` function. The `TSemaphoreBuffer` structure is shown in Listing 5.27 and contains the semaphore you want to perform the operation on, the amount the semaphore's counter should increase or decrease by, and an optional flag which will allow the semaphore counter to be increased if the process terminates. Listing 5.28 demonstrates performing a request to decrease the semaphore's counter on a single semaphore value.

The rules of working with semaphores are quite simple. When you want access to a resource, you decrease the counter by one. If the semaphore counter has a value greater than zero, it has a resource it can allocate to you; the `semop` function will return immediately and the semaphore's counter will be decreased by one. If, on the other hand, the semaphore counter is zero, all of the resources are in use, so the process will actually wait until a resource is available as a result of another process increasing the semaphore counter. Increasing the semaphore counter is just as simple as decreasing the value. You simply set the `TSemaphoreBuffer.sem_op` to a positive value that you want to increase it by.

Listing 5.27 - Using the `TSemaphoreBuffer` data structure

```
type
    sembuf = record
```



```

sem_num: Smallint;
sem_op: Smallint;
sem_flg: Smallint;
end;

TSemaphoreBuffer = sembuf;
PSemaphoreBuffer = ^TSemaphoreBuffer;

```

Listing 5.28 - Performing a semaphore operation

```

var
    operation: TSemaphoreBuffer;

begin
    with operation do
    begin
        sem_num := 0;
        sem_op := -1;           //The value we want to change the semaphore counter by
        sem_flg := SEM_UNDO;    //Release the resource if this process is terminated
    end;

    //Perform the operation
    if semop(SemaphoreHandle, @operation, 1) = -1 then
    begin
        Showmessage('The semaphore operation could not be performed.');
```

```

        Exit;
    end;
end;
```

Listing 5.29 shows a simple application that creates a semaphore and allows you to increase or decrease the counter of the semaphore. By running two or more copies of the application, you can see how the semaphore mechanism works. To see the application working, you can take all the resources that the semaphore has from one of the instances of the application and then request a resource from the same semaphore in the other instance of the application. You will be able to see that the application will wait until a resource is released before the application continues on.

You will also see that you can give back more resources to the semaphore than have been created. This is an important point to note and demonstrates that the use of semaphores is a convention that your applications should use for resource protection and all applications should use the same logic of resource access. If you do not follow the conventions, the use of semaphores to protect resources is not such a good idea.

Figure 5.11 -
The
semaphore
taker
application



Listing 5.29 - Using the semaphore taker application

```

unit unitMainForm;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

type
  TfrmMain = class(TForm)
    Label1: TLabel;
    Label3: TLabel;
    edSemaphoreIPCID: TEdit;
    btnOpen: TButton;
    Label4: TLabel;
    edAmountOfResources: TEdit;
    btnGetSemaphoreCounter: TButton;
    Label2: TLabel;
    btnGetResource: TButton;
    btnReleaseResource: TButton;
    Button1: TButton;
    lblSemaphoreCounter: TLabel;
    procedure btnOpenClick(Sender: TObject);
    procedure btnGetSemaphoreCounterClick(Sender: TObject);
    procedure btnGetResourceClick(Sender: TObject);
    procedure btnReleaseResourceClick(Sender: TObject);
  private
    SemaphoreHandle: integer;
  public
    procedure DisplaySemaphoreInformation;
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.xfm}

uses
  Libc, LibcHelperFunctions, SyncObjs;

procedure TfrmMain.btnOpenClick(Sender: TObject);
var
  sem: TSemun;
begin
  SemaphoreHandle := semget(strtoint(edSemaphoreIPCID.Text),
    strtoint(edAmountOfResources.Text),
    CreatePermissionsMask(upEveryoneCanExecute) or IPC_CREAT);

  if SemaphoreHandle = -1 then
  begin
    Showmessage('The semaphore could not be created.');
```

```

        Exit;
    end;

    //We should now initialize the value of the semaphore
    if semctl(SemaphoreHandle, 0, SETVAL, strtoint(edAmountOfResources.Text)) = -1 then
    begin
        Showmessage('The semaphore could not be initialised.');
```

Exit;

```
    end;

    DisplaySemaphoreInformation;
end;

procedure TfrmMain.DisplaySemaphoreInformation;
var
    ResourcesAvailable: integer;
begin
    //Displays the details of semaphore
    ResourcesAvailable := semctl(SemaphoreHandle, 0, GETVAL);
    if ResourcesAvailable = -1 then
        Showmessage('Oops. We couldn't get the semaphore resource counter')
    else
        lblSemaphoreCounter.Caption := inttostr(ResourcesAvailable);
end;

procedure TfrmMain.btnGetSemaphoreCounterClick(Sender: TObject);
begin
    DisplaySemaphoreInformation;
end;

procedure TfrmMain.btnGetResourceClick(Sender: TObject);
var
    operation: TSemaphoreBuffer;
begin
    //Execute a single semaphore operation
    //The operation in question will be the taking of a resource
    //which will decrease the semaphore's counter by one.

    //Performing this operation will result in the process waiting
    //until a semaphore handle is available.
    with operation do
    begin
        sem_num := 0;
        sem_op := -1;
        sem_flg := SEM_UNDO;
    end;

    //Perform the operation
    if semop(SemaphoreHandle, @operation, 1) = -1 then
    begin
        Showmessage('The semaphore operation could not be performed.');
```

Exit;

```
    end;
```

```

    DisplaySemaphoreInformation;
end;

procedure TfrmMain.btnReleaseResourceClick(Sender: TObject);
var
    operation: TSemaphoreBuffer;
begin
    //Execute a single semaphore operation
    //The operation in question will be the releasing of a resource
    //which will increase the semaphore's counter by one
    with operation do
    begin
        sem_num := 0;
        sem_op := 1;
        sem_flg := SEM_UNDO;
    end;

    //Perform the operation
    if semop(SemaphoreHandle, @operation, 1) = -1 then
    begin
        Showmessage('The semaphore operation could not be performed.');
```

Exit;

```

    end;

    DisplaySemaphoreInformation;
end;

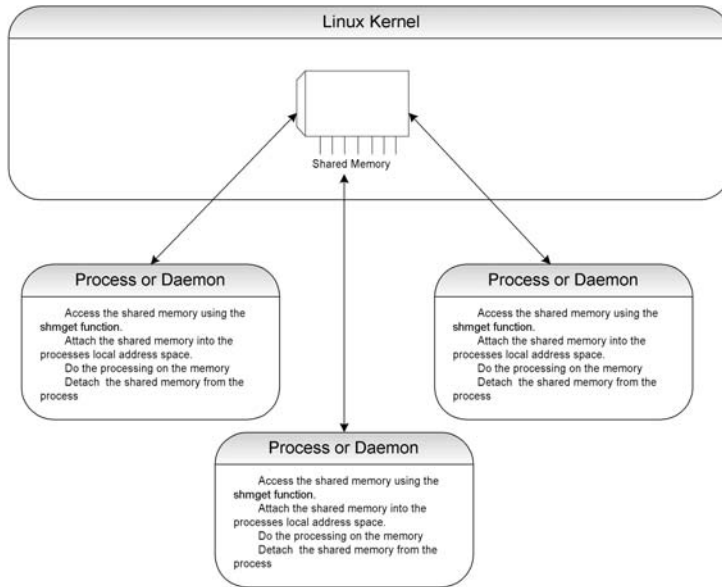
end.
```

So you've seen the benefits of using semaphores in your application to protect resources. Now let's make use of the semaphore by protecting another interprocess method — shared memory.

Shared Memory

Shared memory allows different processes to reference a block of memory allocated by the kernel so that unrelated processes can access the memory within the process's own address space. Of all the interprocess communication methods, this method is the most efficient, and like semaphores and message queues, it has a simple API to utilize the communication method.

Figure 5.12 -
Shared
memory



Shared memory works by a process requesting that the kernel either create a block of memory of a particular size or obtain the handle of a previously created block of memory. After this memory allocation is done and a handle is obtained to the memory, processes request that the shared memory created by the kernel be attached to the current process's memory space. Once that operation is performed, then accessing the shared memory data is as easy as working with the pointer that is returned after you have attached to the memory.

The data is yours to play with and do as you wish. You can place large amounts of text data in there, database records, or a combination of whatever you want. Dealing with this memory is simple a matter of knowing how to use pointers within your application. That is all there is to it.

Once you have completed your operations on the shared memory, your process can detach from the shared memory. Doing this will cause the shared memory to be no longer mapped into your process's memory space. Like message queues and semaphores, the shared memory API also has a function to perform special control operations on the shared memory such as obtaining the details and removing the interprocess communication method.

Programming Shared Memory

The shared memory API like the other interprocess communication methods has a very small API. In fact only four functions are used to make use of shared memory. The functions are shown in Listing 5.30.

Listing 5.30 - Using the shared memory functions

```

function shmget(__key: key_t; __size: size_t; __shmflg: Integer): Integer; cdecl;
function shmat(__shmid: Integer; __shmaddr: Pointer; __shmflg: Integer): Pointer;
               cdecl;
  
```

```
function shmdt(__shmaddr: Pointer): Integer; cdecl;
function shmctl(__shmid: Integer; __cmd: Integer;
    __buf: PSharedMemIdDescriptor): Integer; cdecl;
```

The first of the functions, `shmget`, will create a shared memory block, which is `__size` bytes large. The `__shmflg` option specifies which users can access the shared memory, the same as when you create a file with the `creat` function. You also have the option of including the `IPC_CREAT` constant to make sure that the shared memory is created if it does not exist. The code to create the shared memory is shown in Listing 5.31. Calling this function will either create access to the shared memory and return the shared memory handle or return the handle of a previously created handle.

Listing 5.31 - Creating a shared memory handle

```
SharedMemHandle := shmget(SHAREDMEMORYID,
    sizeof(boolean) * 7, CreatePermissionsMask(upAllPermissions) or IPC_CREAT);

if SharedMemHandle = -1 then
begin
    Showmessage('Oops... The Shared Memory could not be created.');
```

```
    Exit;
end;
```

Once the handle has been obtained from a successful call to `shmget`, then to attach to the shared memory within your calling application's process space simply call the `shmat` function. Calling this function results in the shared memory being mapped to the process space is as easy as using the code in Listing 5.32.

Listing 5.32 - Attaching and detaching from shared memory

```
LocalPointer := shmat(SharedMemHandle, nil, 0);

if LocalPointer = nil then
begin
    Showmessage('Oh no. We could not attach to the shared memory.');
```

```
    Exit;
end;
```

```
//Access the shared memory by using the LocalPointer
//..
```

```
//Lets detach from the Shared Memory
shmdt(LocalPointer);
```

As well as attaching the shared memory to your application, it is also worth detaching from your application when you are done. Doing this is a matter of calling the `shmdt` function shown in Listing 5.32 and passing the pointer returned from a previous call to `shmat`.

Like other methods of interprocess communication, shared memory also has a function that you can use to do specific operations such as read or modify the access permissions of the shared memory and to tell the kernel that the interprocess communication method is no longer required. This is done by using the `shmctl` function. Using this function, as shown in Listing 5.33, you specify the shared memory handle, the command that you want to operate, and any additional parameters that the specific command you are using requires such as the permissions that you are reading and the permissions that are being set.

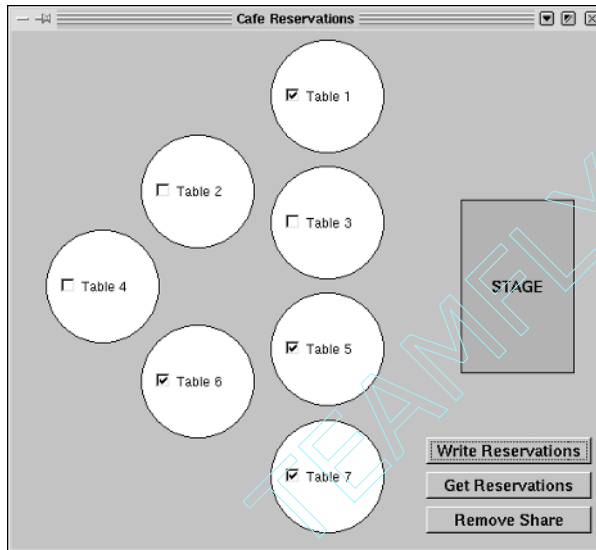
Listing 5.33 - Using the shmctl function

```
//Tell the kernel to remove the Shared Memory that we allocated.
shmctl(SharedMemHandle, IPC_RMID, nil);
```

Using Semaphores with Shared Memory

Before shared memory, we looked at the benefits of using semaphores as a way of flagging when a resource is in use. So it makes sense to use semaphores when your applications use shared memory, so that you can be sure that you do not have two processes that are accessing the same memory block and modifying the memory in such a way as to make the data in the memory block inconsistent.

*Figure 5.13 -
The cafe
reservation
application*



In our example for shared memory, we will look at a simple reservation system for a cafe or restaurant shown in Figure 5.13. The purpose of this application is to make bookings for customers so that they can choose their table.

The resource to protect in this application are the tables. The most obvious problem that can occur in a scenario such as this is that a table can be double-booked. As a result, you should ensure that entries cannot be written to the tables at the same time by using a semaphore to protect when entries are written to the shared memory. You should note that doing this is only a convention for processes that access the shared memory. You could easily write a separate application that does what it likes with the shared memory. This would be extremely bad for the stability of your application as the data could be in a completely inconsistent state. In this case, if any rogue piece of software accesses your application, it will be shot to bits.

This is why it is extremely important to make sure that all processes access the data using the same convention. You should try to adopt a common solution for all of these issues, such as creating a Kylix class that handles all communication in a single class and forcing all applications to use this unit to ensure that the data is correctly managed.

In this example of the application, we will use a single semaphore to control the process of writing the reservations to the shared memory. In addition, before an entry is added, we will access the shared memory to see if any further reservations have been made. This is our safety net for the application. A further extension of this example is that you may have several semaphores, each controlling access to a particular table, but for the purposes of this example, a single semaphore should be enough.

Listing 5.34 - Using the cafe reservation application

```
unit unitMainForm;

interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
    QStdCtrls, QButtons, QExtCtrls, QComCtrls, Libc, LibcHelperFunctions;

type
    TfrmMain = class(TForm)
        btnWriteReservations: TButton;
        btnGetReservations: TButton;
        btnClose: TButton;
        Shape1: TShape;
        Shape2: TShape;
        Shape3: TShape;
        Shape4: TShape;
        Shape5: TShape;
        Shape6: TShape;
        Shape7: TShape;
        cbTable1: TCheckBox;
        cbTable2: TCheckBox;
        cbTable4: TCheckBox;
        cbTable3: TCheckBox;
        cbTable5: TCheckBox;
        cbTable6: TCheckBox;
        cbTable7: TCheckBox;
        shpStage: TShape;
        Label1: TLabel;
        procedure FormCreate(Sender: TObject);
        procedure btnCloseClick(Sender: TObject);
        procedure btnGetReservationsClick(Sender: TObject);
        procedure btnWriteReservationsClick(Sender: TObject);
        procedure DoubleCheckTable(Sender: TObject);
    private
        function GetTableCheckBox(Index: integer): TCheckBox;
        function GetTableIndex(ACheckBox: TCheckBox): integer;
    public
        SharedMemHandle: integer;
        SemaphoreHandle: integer;
        LoadingTableInfo: boolean;
        procedure OpenSharedMemory;
        procedure SaveTableAllocations;
        procedure LoadAllocationsFromSharedMemory;
        property TableCheckBox[Index: integer]: TCheckBox read GetTableCheckBox;
    end;
```



```

var
    frmMain: TfrmMain;

implementation

{$R *.xfm}

Const
    SHAREDMEMORYID = 2231;
    SEMAPHOREID = 2232;

type
    PCafeTableAllocations = ^TCafeTableAllocations;
    TCafeTableAllocations = array[1..7] of boolean;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
    //Allocate the Shared Memory
    OpenSharedMemory;
    LoadAllocationsFromSharedMemory;
end;

procedure TfrmMain.LoadAllocationsFromSharedMemory;
var
    Reservations: PCafeTableAllocations;
    counter: integer;
begin
    LoadingTableInfo := true;

    //Here we map the shared memory into the address
    //space so we can load the data.

    //Let's attach to the shared memory
    Reservations := shmat(SharedMemHandle, nil, 0);

    if Reservations = nil then
    begin
        Showmessage('Oh no. We could not attach to the shared memory.');
```

Exit;

```

    end;

    //Let's display the information
    for counter := 1 to 7 do
        TableCheckBox[counter].Checked := Reservations^[counter];

    //Let's detach the Shared Memory
    shmdt(Reservations);

    LoadingTableInfo := false;
end;

procedure TfrmMain.OpenSharedMemory;
begin
    SharedMemHandle := shmget(SHAREDMEMORYID,
        sizeof(boolean) * 7, CreatePermissionsMask(upAllPermissions) or IPC_CREAT);
```

```

if SharedMemHandle = -1 then
begin
    Showmessage('Oops... The Shared Memory could not be created.');
```

Exit;

```
end;
```

//Let's manage the Shared Memory with a semaphore

```
SemaphoreHandle := semget(SEMAPHOREID, 1,
    CreatePermissionsMask(upAllPermissions) or IPC_CREAT);
```

```
if SemaphoreHandle = -1 then
begin
    Showmessage('Oops... The Semaphore could not be created.');
```

Exit;

```
end;
```

//Initialize the Semaphore

```
if semctl(SemaphoreHandle, 0, SETVAL, 1) = -1 then
    Showmessage('The semaphore could not be initialised.');
```

end;

```
procedure TfrmMain.SaveTableAllocations;
var
    counter: integer;
    Reservations: PCafeTableAllocations;
    op: TSemaphoreBuffer;
begin
    //Before we even attempt to save the data, we will make sure
    //that some one else is not writing to the system by
    //using a semaphore
    with op do
    begin
        sem_num := 0;
        sem_op := -1;
        sem_flg := SEM_UNDO;
```

end;

//This function will not return unless it has the resource.

```
semop(SemaphoreHandle, @op, 1);
```

//Here we map the shared memory into the address

//space so we can Save the data

//attach to the shared memory

```
Reservations := shmat(SharedMemHandle, nil, 0);
```

```
if Reservations = nil then
begin
    Showmessage('Oh no. We could not attach to the shared memory.');
```

Exit;

```
end;
```

//Let's save the information

```
for counter := 1 to 7 do
    Reservations^[counter] := TableCheckBox[counter].Checked;
```

```

//Let's detach the Shared Memory
shmdt(Reservations);

//We should allow other user to use the semaphore so that they can
//write to the journal
with op do
begin
    sem_num := 0;
    sem_op := 1;
    sem_flg := SEM_UNDO;
end;

semop(SemaphoreHandle, @op, 1);
end;

procedure TfrmMain.btnCloseClick(Sender: TObject);
begin
    //Get rid of the shared memory
    shmctl(SharedMemHandle, IPC_RMID, nil);

    //Release the Semaphore Handle
    semctl(SemaphoreHandle, 0, IPC_RMID);
end;

procedure TfrmMain.btnGetReservationsClick(Sender: TObject);
begin
    LoadAllocationsFromSharedMemory;
end;

procedure TfrmMain.btnWriteReservationsClick(Sender: TObject);
begin
    SaveTableAllocations;
end;

function TfrmMain.GetTableCheckBox(Index: integer): TCheckBox;
begin
    case Index of
        1: Result := cbTable1;
        2: Result := cbTable2;
        3: Result := cbTable3;
        4: Result := cbTable4;
        5: Result := cbTable5;
        6: Result := cbTable6;
        7: Result := cbTable7;
    else
        Result := nil;
    end;
end;

function TfrmMain.GetTableIndex(ACheckBox: TCheckBox): integer;
var
    i: integer;
begin
    Result := -1;

    for i := 1 to 7 do
        if ACheckBox = GetTableCheckBox(i) then

```

```

        begin
            Result := i;
            break;
        end;
    end;

procedure TfrmMain.DoubleCheckTable(Sender: TObject);
var
    TablePosition: integer;
    Reservations: PCafeTableAllocations;
begin
    //This function will double check to see if a table has
    //already been in the table when the user clicks on it.
    //If it has then set the check-box to be checked
    //and display a warning message.
    TablePosition := GetTableIndex(Sender as TCheckBox);

    if (Sender as TCheckBox).Checked then
    begin
        //attach to the shared memory
        Reservations := shmat(SharedMemHandle, nil, 0);

        if Reservations = nil then
        begin
            Showmessage('Oh no. We could not attach to the shared memory.');
```

Exit;

```

        end;

        //Let's see if the table is already booked
        if (not LoadingTableInfo) and Reservations^[TablePosition] then
            if MessageDlg('The table may be already booked. Are you sure you want to '+
                'Double Book this one. It is not recommended.', mtConfirmation,
                [mbYes, mbNo], 0) = mrNo then
                (Sender as TCheckBox).Checked := false;

        //Let's detach the Shared Memory
        shmdt(Reservations);
    end;
end;

end.
```

Creating Unique IPC Keys

Message queues, semaphores, and shared memory resources are stored within the Linux kernel and are accessible from multiple processes. As a result, the processes that request the IPC resource need a method of uniquely identifying the IPC resource that they want to use. The identifier that is used is called the IPC key, which is simply an integer identifier.

When a function such as `msgget`, `shmget`, or `semget` is used to create one of these IPC resources, all of the functions expect an IPC key as the first parameter to state which IPC resource they want to access. The second parameter is a flag that includes the permissions for the IPC resource and indicates if the resource should be created.

There are a few options available to you for this IPC key. The first is that you can use the `IPC_PRIVATE` constant, which will automatically create a unique key for you. Once

you have the handle number of the resource, however, you will need to fork the process to allow other processes to have the number or save the number to a file, such as in the `/etc/myapp/config.txt` file and make other processes read the key value in that file when they open the IPC resource.

The second option is that you can pick a number and use that as the IPC key. In some cases you may want to do this; however, you may have other applications that may use the same key. If one application is using that key for different purposes, your application may end up in severe trouble. Although this solution may work in a controlled environment, it is not the safest of methods.

The last option is to use the `ftok` function, which takes the name of an existing file and a project number which is between 0 and 255 to make a valid unique IPC key. If you have several different types of applications that use the same configuration file, for example, you can easily use the configuration file as the parameter. The second number is simply your preference for a number. This is a solid solution to the problem of creating a unique IPC key, especially if all your applications live in the same directory. In this way you can use code similar to the code in Listing 5.35.

Listing 5.35 - Creating a unique IPC key

```
var
    MyHandle: integer;

...
    MyHandle := ftok(
        PChar(extractFilePath(Application.EXENAME)+'config.ini'), 25);
```

Which Methods Work Best for Different Kinds of Applications

Now that you've examined some of the different types of interprocess communication that you will want to use, you know that signals are not really effective as a method of communication of data, just of simple tokens. If you want a communication method that will work if your application is an X or Qt application, including CLX, you will not be able to use pipes. Pipes and named pipes are not the most effective way to work with records.

If you are working with chunks of data, such as records, and need a method of interprocess communication, message queues or shared memory would be the best choices because their respective APIs allow the ability to obtain a chunk of data at a given time. Message queues and shared memory also have the ability to have their data come from many different sources without too many problems. Pipes and named pipes would require much more programming to set up a system where multiple processes could be writing to one end of the pipe.

Although semaphores are classed as an interprocess communication method, they do not communicate data, but rather the status of a particular resource between processes. As a result, you would only need to use semaphores when access to a resource needs to be restricted between processes.

API Reference

__raise *LibC.pas*

Syntax

```
function __raise(
  SigNum: Integer;
):Integer;
```

Description

The `__raise` function sends a signal to the current process. Calling the `__raise` function is the same as calling the `kill` function by passing in the result of `getpid()` as the process identifier.

Parameters

`SigNum`: This is the signal that will be passed to the process.

Return Value

The function returns 0 if successful and -1 if the function failed.

See Also

`kill`, `signal`

Example

Listing 5.36 - Using the `__raise` function

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //Sending the SIGKILL process to this app
  //will terminate it.
  __raise(SIGKILL);
end;
```

alarm *LibC.pas*

Syntax

```
function alarm(
  Seconds: Cardinal;
):Cardinal;
```

Description

The `alarm` function will send the `SIGALRM` signal to the current process after a delay of a specified number of seconds.

Parameters

`Seconds`: This is the number of seconds that should expire before the current process receives the `SIGALRM` signal.

Return Value

The alarm function will return 0 if no other alarm was set for the process. Otherwise, the alarm function will return the number of seconds until the process receives an alarm notification.

See Also

kill, __raise, signal

Example

Listing 5.37 - Using the alarm function

```
procedure AlarmSignalGrabber(Signal: integer); cdecl;
begin
    //We will get here when we get the alarm
    frmAlarmStatus.lblStatus.Caption := 'Got the alarm.';
end;

procedure TfrmAlarmStatus.btnSendAlarmClick(Sender: TObject);
begin
    //Send the Alarm to this process
    alarm(seSeconds.Value);
end;

procedure TfrmAlarmStatus.FormCreate(Sender: TObject);
begin
    //Set up that all alarms (SIGALRM) are handled by the
    //AlarmSignalGrabber function
    signal(SIGALRM, AlarmSignalGrabber);
end;
```

kill LibC.pas

Syntax

```
function kill(
    ProcessID: __pid_t;
    SigNum: Integer
):Integer;
```

Description

The kill function will send a signal to a particular process. If the process that the signal being sent to has not set up a method of retrieving the signal, the default behavior is for the process to terminate.

Parameters

ProcessID: This is the process that the signal will be sent to.

SigNum: This is the signal number that will be sent to the process. This value is normally a value from the signal list in Listing 5.1.

Return Value

If successful, the function returns 0. If the function was unsuccessful, it returns -1. The function call will normally fail if the signal that is sent is not a valid signal, the process does not exist, or the process calling the function does not have access to send the signal to the process.

See Also

`__raise`, `signal`, `killpg`

Example

See Listing 5.4 - Sending a signal to a process.

killpg LibC.pas**Syntax**

```
function killpg(
  ProcessGrp: __pid_t;
  SigNum: Integer
):Integer;
```

Description

This function is similar to the `kill` function, but instead of sending the signal to a process, it sends the signal to the process group.

Parameters

ProcessGrp: This is the process group to which to send the signal. If this parameter is 0, the signal will be sent to this process group.

SigNum: This is the signal number that will be sent to the process group. This value is normally a value from the signal list in Listing 5.1.

Return Value

When successful, this function returns 0. When unsuccessful, the function returns -1. The function call will normally fail if the signal that is sent is not a valid signal, the process group does not exist, or the process calling the function does not have access to send the signal to the process.

See Also

`kill`

Example**Listing 5.38 - Using the killpg function**

```
procedure TfrmProcessDemo.TerminateProcessGroup;
begin
  //This sends the kill signal to the processes, process group
```



```
    killpg(0, SIGKILL);
end;
```

mkfifo *LibC.pas*

Syntax

```
function mkfifo(
  PathName: PChar;
  Mode: __mode_t;
):Integer;
```

Description

This function will create a named pipe (fifo) which is a special type of file designed to allow processes to communicate between themselves by using a named identifier to access the pipe. Once the pipe has been created with the mkfifo function, when opening the file for read access, you will open the reading end of the pipe. Likewise, when opening the file with write access, the writing end of the pipe will be returned.

Parameters

PathName: This is the path where the named pipe will be created. You should allow processes that wish to use the named pipe to access the named pipe using the same value passed in here.

Mode: Mode defines the user permissions for the named pipes. This is the same value that is used for the creat function. For convenience, it is best to use the CreatePermissionsMask function found in the LibCHelperFunctions unit on the companion CD-ROM.

Return Value

When successful, this function returns 0. When unsuccessful, the function returns -1. The main causes of the function failing are if the calling process does not have write access to the directory where the fifo is being created, or the fifo already exists.

See Also

open, __close, popen, pclose

Example

Listing 5.39 - Using the mkfifo function

```
var
  i: integer;
begin
  //Let's create the named pipe
  i := mkfifo(PChar(extractFilePath(Application.EXENAME)+'TempFifo'), 666);
  if i = -1 then
    showmessage('Could not create the named pipe.');
```

end;

msgctl LibC.pas**Syntax**

```
function msgctl(
  __msqid: Integer;
  __cmd: Integer;
  __buf: PMsgQueueIdDesc;
):Integer;
```

Description

The msgctl function performs a specific command on a valid message queue, such as obtaining the properties of the message queue, changing the permissions of the message queue, and removing the message queue resource.

Parameters

__msqid: This is the handle to a valid message queue returned from a successful call to msgget.

__cmd: This is the command that is to be executed, which will be either IPC_STAT, IPC_SET, or IPC_RMID. Using IPC_STAT for this value will result in the properties being copied to the TMsgQueueIdDesc record in the __buf parameter. If the command is IPC_SET, the msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes attributes of the message queue will be changed to the data that is in the __buf parameter. If this value is IPC_RMID, the shared memory resource will be removed immediately. Only the creator of the message queue, the owner of the message queue, or a superuser has permissions to remove a message queue.

__buf: For the IPC_STAT and IPC_SET commands, this parameter points to the buffer that receives or sets the buffer operations, respectively.

Return Value

If the function is successful, it returns 0. If the function fails, it returns -1.

See Also

msgget, msgrcv, msgsnd

Listing 5.40 - Using the TMsgQueueIdDesc data structure and the msgctl function

```
type
{ Data structure used to pass permission information to IPC operations. }
type
  ipc_perm = record
    __key: __key_t;           { Key. }
    uid: __uid_t;             { Owner's user ID. }
    gid: __gid_t;             { Owner's group ID. }
    cuid: __uid_t;            { Creator's user ID. }
    cgid: __gid_t;            { Creator's group ID. }
    mode: Smallint;           { Read/write permission. }
    __pad1: Smallint;
    __seq: Smallint;          { Sequence number. }
```

```

    __pad2: Smallint;
    __unused1: LongWord;
    __unused2: LongWord;
end;
TIpCPermission = ipc_perm;
PIpCPermission = ^TIpCPermission;

msqid_ds = record
    msg_perm: ipc_perm;           { structure describing operation permission }
    msg_stime: __time_t;          { time of last msgsnd command }
    __unused1: LongWord;
    msg_rtime: __time_t;          { time of last msgrcv command }
    __unused2: LongWord;
    msg_ctime: __time_t;          { time of last change }
    __unused3: LongWord;
    __msg_cbytes: LongWord;       { current number of bytes on queue }
    msg_qnum: msgqnum_t;          { number of messages currently on queue }
    msg_qbytes: msglen_t;         { max number of bytes allowed on queue }
    msg_lspid: __pid_t;           { pid of last msgsnd() }
    msg_lrpid: __pid_t;           { pid of last msgrcv() }
    __unused4: LongWord;
    __unused5: LongWord;
end;

TMsgQueueIdDesc = msqid_ds;
PMsgQueueIdDesc = ^TMsgQueueIdDesc;

//To remove a message queue

msgctl(queueHandle, IPC_RMID, nil);

```

msgget LibC.pas

Syntax

```

function msgget(
    __key: key_t;
    __msgflg: Integer;
):Integer;

```

Description

The msgget function returns a handle to a message queue either by returning a message queue that has been created or creating a message queue and returning the new handle of the message queue.

Parameters

__key: This is a valid IPC key which will be a predefined number, the IPC_PRIVATE constant, which is used when creating a new message queue, or a value taken from a call to ftok.

__msgflg: This parameter sets the permissions of who is able to access this resource. Using the IPC_CREAT option in this flag will create the message queue if it hasn't been created

already. Using the `IPC_EXCL` option in the function will result in the function failing if the message queue already exists.

Return Value

When successful, the function returns a positive integer handle to a message queue that can be used in the `msgctl`, `msgsnd`, and `msgrcv` functions. If the function fails, the return value is `-1`.

See Also

`msgctl`, `msgsnd`, `msgrcv`

Example

Listing 5.41 - Using the `msgget` function

```
procedure TfrmSimpleMessage.CreateMessageQueue;
var
  FIPCKey: integer;
begin
  FIPCKey := ftok(PChar(Application.EXENAME), 2);
  FQueueID := msgget(FIPCKey, 600 or IPC_CREAT);
  if FQueueID = -1 then
    begin
      Showmessage('The Message Queue could not be created.');
```

```
    end;
```

```
end;
```

msgrcv ***LibC.pas***

Syntax

```
function msgrcv(
  __msqid: Integer;
  var Msg;
  __msgsz: size_t;
  __msgtyp: Longint;
  __msgflg: Integer
):Integer;
```

Description

This function retrieves a message from a message queue. If the message queue is empty, the function will wait until a message is sent to the message queue before the function returns. This function gives the option of retrieving the next message in the queue, the next message of a specific priority, the next message with the highest priority, or the next message regardless of priority.

Parameters

`__msqid`: This is the handle of the message queue returned from a successful call to `msgget`.

`Msg`: This is a pointer to a buffer that will contain the message that is received.

`__msgsz`: This is the size of the message that will be retrieved. The size of the message does not include the longint priority value.

`__msgtyp`: This option specifies which message priority will be received. If this value is 0 the next message of any priority will be retrieved using a first-in, first-out rule. If this value is a number greater than 0, the message of that specific priority will be reached. If the message is less than 0, the next message on the queue less than the absolute value of this parameter will be retrieved.

`__msgflg`: This parameter contains the options for the retrieval of the message. If the value includes `IPC_NOWAIT`, then the function will return immediately if there are no messages that it can retrieve. If this value includes `MSG_NOERROR`, and the message that is in the queue is larger than the value in `__msgsz`, the message will be truncated to fit the local size. If this parameter includes `MSG_EXCEPT` and the message priority type in the `__msgtyp` parameter is positive, the message returned will be any message type except the one specified in `__msgtype`.

Return Value

The function returns the number of bytes copied into the message when successful and `-1` if the function fails.

See Also

`msgsnd`, `msgget`, `msgctl`

Example

See Listing 5.21 - The message queue server application.

msgsnd LibC.pas

Syntax

```
function msgsnd(
  __msqid: Integer;
  const Msgp;
  const __msgsz: size_t;
  __msgflg: Integer
):Integer;
```

Description

The `msgsnd` function adds a message to an open message queue.

Parameters

`__msqid`: This is the handle of the message queue returned from a successful call to `msgget`.

`Msgp`: This is the pointer to some message structure that will be sent to the message queue. The first part of this message should include a priority of the message.

`__msgsz`: This is the size of the message that will be sent to the message queue minus the size of the priority field (longint) in the message.

`__msgflg`: This parameter contains the options for message retrieval. See the `msgrcv` function for these values.

Return Value

This function returns 0 when successful and -1 when the function fails.

See Also

`msgrcv`, `msgget`, `msgctl`

Example

See Listing 5.22 - The message queue client application.

pause LibC.pas

Syntax

```
function pause:Integer;
```

Description

The `pause` function will cause the calling process to wait until a signal is received by the process.

Return Value

The `pause` function will always return -1.

See Also

`__sigpause`

Example

See Listing 5.3 - Catching the SIGINT signal from a console application.

pclose LibC.pas

Syntax

```
function pclose(
  Stream: PIOFile
):Integer;
```

Description

The `pclose` function closes an input/output stream that was opened with the `popen` function. Although an input output stream is returned by the `popen` function, closing the stream with the `__close` function is not sufficient to close the stream; it must be closed with the `pclose` function.

Parameters

Stream: This is the stream that was returned by a call to `popen`.

Return Value

If successful, the function returns 0. If the function failed, it returns `-1`.

See Also

`popen`, `pipe`

Example

See Listing 5.42 - Using the `popen` and `pclose` functions.

pipe* LibC.pas*Syntax**

```
function pipe(
  PipeDes: PInteger;
):Integer;

function pipe(
  var PipeDes: TPipeDescriptors;
):Integer;
```

Description

The `pipe` function creates a pipe where data will be written in one end and read from the other. The read and write ends of the pipe are returned as two file descriptors. Although this function is overloaded, what is placed in the `PipeDes` parameters are two file descriptors — the first is the read file descriptor of the pipe and the second is the write file descriptor for the pipe.

Parameters

`PipeDes`: This parameter is either a pointer to an array of integers holding at least two elements or a variable reference to a `TPipeDescriptors` data structure.

```
TPipeDescriptors = record
  ReadDes: Integer;
  WriteDes: Integer;
end;
```

Return Value

When successful, this function returns 0. When unsuccessful, the function returns `-1`.

See Also

`fork`, `__close`, `popen`, `pclose`

Example

See Listing 5.14 - A simple program that makes use of pipes.

popen* LibC.pas**Syntax***

```
function popen(
  const Command: PChar;
  Modes: PChar;
):PIOFile;
```

Description

The popen function creates a pipe within the current process and then forks the current process, executes a shell command, and attaches either the standard input or standard output of the created process, depending on whether the calling process wants to send data to the new process or read the information from the created process.

Parameters

Command: This is the Linux command to execute that is passed to the Linux shell.

Modes: This string parameter specifies if the file should be open for reading or writing. For reading, the r value should be used, and for writing, the w should be used. All other values will result in the function failing.

Return Value

When successful, the function returns a valid input/output stream. If the function is unsuccessful, it returns nil.

See Also

pipe, pclose

Example**Listing 5.42 - Using the popen and pclose functions**

```
program Usingpopen;

(*
   This function demonstrates using the popen function
   to obtain the information from a separate application
*)

{$APPTYPE CONSOLE}

uses
  Libc;

var
  AppStream: PIOFile;
  Line: array[0..500] of char;
  intCharsRead: integer;

begin
  writeln('Let's analyze a hard drive on the system.');
```



```

AppStream := popen('ide_info /dev/hda', 'r');
try
  if AppStream = nil then
  begin
    writeln('The command could not be executed.');
```

```

  end else begin
    //If we are here that means that we have a link into
    //standard output of the newly created process.
    while feof(AppStream) = 0 do
    begin
      fgets(Line, 500, AppStream);
      Writeln(Line);
    end;
  end;
finally
  if AppStream <> nil then
    pclose(AppStream);
end;
end.
```

psignal LibC.pas

Syntax

```

procedure psignal(
  SigNum: Integer;
  const S: PChar
);
```

Description

The `psignal` function writes the details about the signal passed into the `SigNum` parameter out to standard error, followed by a colon, a space, and the null-terminated string that was passed as the `S` parameter.

Parameters

SigNum: This is the signal number that will be used to determine a description of the signal number, such as `SIGWAIT`.

S: This is the text message that will be displayed before a colon and the descriptive name of the signal on standard error.

Example

Listing 5.43 - Using the `psignal` function

```

program psignalDemo;

{$APPTYPE CONSOLE}

uses
  Libc;

begin
  writeln('Writing out the descriptions of some signals.');
```

```

psignal(SIGKILL, 'SIGKILL');
psignal(SIGALRM, 'SIGALRM');
psignal(SIGCHLD, 'SIGCHLD');
end.

```

semctl LibC.pas

Syntax

```

function semctl(
  __semid: Integer;
  __semnum: Integer;
  __cmd: Integer;
):Integer; cdecl; varargs;

```

Description

The `semctl` function performs housekeeping tasks on a semaphore including obtaining or setting permission details about the semaphore, initializing the value of one or more semaphores, and telling the kernel to remove the semaphore.

Parameters

`__semid`: This is the semaphore ID that was returned by a successful call to `semget`.

`__semnum`: This is the semaphore to perform the command operation on. For a binary semaphore, this value is normally 0; otherwise, this value should be the zero-based index of the semaphore that you are working on.

`__cmd`: This is the command to perform on the semaphore and will be one of the following values.

`IPC_STAT`: This command retrieves permission information about the semaphore into a `TSemaphoreIdDescriptor` buffer, which should be the first `varargs` parameter for the function.

`IPC_SET`: This command will set the permission access for the semaphore using the `TSemaphoreIdDescriptor`. However, the only fields that will be changed are the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields. A pointer to a `TSemaphoreIdDescriptor` should be the first `varargs` parameter when using this command.

`IPC_RMID`: This command removes the semaphore from the kernel.

`GETALL`: This will return all the semaphore values into an array of integer elements. A pointer to an integer array should be the first `varargs` parameter of this function.

`SETALL`: Using this command will set the values of the semaphore. The size of the array of integers should be the same number of elements when the semaphore was created using `semget`. A pointer to an integer array should be the first `varargs` parameter containing the values to set when using this command.

`_GETPID`: This command will return the process ID of the last process that made a call to the `semop` function using this semaphore.

`GETVAL`: Using this command results in the value of a particular semaphore value being returned by the function.

SETVAL: Using this command allows for the semaphore at position __semnum to be set to the value passed into the first varargs parameter which is an integer.

GETNCNT: This command will return how many processes are waiting until a resource is available on this particular semaphore.

GETZCNT: This command will return the number of processes that are waiting until there are no resources available on this semaphore.

Return Value

The function will return -1 if it was unsuccessful. For a successful command, the result depends on the command that was used. For _GETPID, the function will return the process ID. For GETNCNT and GETZCNT the function will return the number of processes waiting and for GETVAL, the function will return the value of the semaphore.

See Also

semget, semop

Example

Listing 5.44 - Using the semctl function

```
procedure TfrmSemctlDemo.btnCreateTheSemaphoreClick(Sender: TObject);
var
  SemKey: integer;
begin
  //Create the Semaphore
  SemKey := ftok(PChar(Application.EXEName), 1);
  SemaphoreID := semget(SemKey, 1, 666 or IPC_CREAT);

  if SemaphoreID = -1 then
    mmInfo.Lines.Add('The Semaphore could not be created/accessed. ');
end;

procedure TfrmSemctlDemo.btnRemoveClick(Sender: TObject);
begin
  //Remove the semaphore.
  semctl(SemaphoreID, 0, IPC_RMID);
end;

procedure TfrmSemctlDemo.btnGetPermissionsClick(Sender: TObject);
var
  buf: TSemaphoreIdDescriptor;
  intRes: integer;
begin
  intRes := semctl(SemaphoreID, 0, IPC_STAT, @buf);
  if intRes <> -1 then
    begin
      //We have information on the semaphore
      mmInfo.Lines.Clear;
      mmInfo.Lines.Add('Permission Information. ');
      mmInfo.Lines.Add('User ID: ' + inttostr(buf.sem_perm.uid));
      mmInfo.Lines.Add('Group ID: ' + inttostr(buf.sem_perm.gid));
      mmInfo.Lines.Add('Creator User ID: ' + inttostr(buf.sem_perm.cuid));
      mmInfo.Lines.Add('Creator ID: ' + inttostr(buf.sem_perm.cgid));
```

```

        mmInfo.Lines.Add('Permissions Mode: '+ inttostr(buf.sem_perm.mode));
    end else
        mmInfo.Lines.Text := 'Could not obtain permission information.';
    end;

procedure TfrmSemctlDemo.btnGetSemaphoreValueClick(Sender: TObject);
var
    SemValue: integer;
begin
    SemValue := semctl(SemaphoreID, 0, GETVAL);
    mmInfo.Lines.Add('The Semaphore Value is '+inttostr(SemValue));
end;

procedure TfrmSemctlDemo.btnSetSemaphoreValueClick(Sender: TObject);
var
    SemValue: integer;
begin
    SemValue := seSetSemaphore.Value;
    semctl(SemaphoreID, 0, SETVAL, SemValue);
end;

procedure TfrmSemctlDemo.btnObtainSemaphoreResourceClick(Sender: TObject);
var
    operation: TSemaphoreBuffer;
begin
    with operation do
    begin
        sem_num := 0;
        sem_op := -1; //The value we want to change the semaphore counter by
        sem_flg := SEM_UNDO; //Release the resource if this process is terminated
    end;

    //Perform the operation
    if semop(SemaphoreID, @operation, 1) = -1 then
        mmInfo.Lines.Add('The semaphore operation could not be performed.');
```

end;

```

procedure TfrmSemctlDemo.btnReleaseSemaphoreResourceClick(Sender: TObject);
var
    operation: TSemaphoreBuffer;
begin
    with operation do
    begin
        sem_num := 0;
        sem_op := 1; //The value we want to change the semaphore counter by
        sem_flg := SEM_UNDO; //Release the resource if this process is terminated
    end;

    //Perform the operation
    if semop(SemaphoreID, @operation, 1) = -1 then
        mmInfo.Lines.Add('The semaphore operation could not be performed.');
```

end;

```

procedure TfrmSemctlDemo.btnSetAllSemaphoreValuesClick(Sender: TObject);
var
    //The array that is passed when using the SETALL
    //must contain the number of elements for the semaphore
```

```

    ValuesToUse: array[0..0] of integer;
begin
    //Although this is a binary semaphore, this demonstrates
    //using the SETALL command.
    ValuesToUse[0] := seSetSemaphore.Value;
    semctl(SemaphoreID, 0, SETALL, @ValuesToUse);
end;

procedure TfrmSemctlDemo.btnGetLastSemOpProcessIDClick(Sender: TObject);
var
    ProcessID: integer;
begin
    ProcessID := semctl(SemaphoreID, 0, _GETPID);
    mmInfo.Lines.Add('The Process ID is '+ inttostr(ProcessID));
end;

procedure TfrmSemctlDemo.ProcessWaitingUntilSemValueIsZeroClick(Sender: TObject);
var
    ProcessesWithSemResources: integer;
begin
    ProcessesWithSemResources := semctl(SemaphoreID, 0, GETZCNT);
    mmInfo.Lines.Add('The number of processes with semaphore resources is '+
        inttostr(ProcessesWithSemResources));
end;

procedure TfrmSemctlDemo.GetProcessesWaitingForResourceClick(
    Sender: TObject);
var
    ProcessesWaitingOnSemResources: integer;
begin
    ProcessesWaitingOnSemResources := semctl(SemaphoreID, 0, GETNCNT);
    mmInfo.Lines.Add('The number of processes waiting for a semaphore resources is '+
        inttostr(ProcessesWaitingOnSemResources));
end;

procedure TfrmSemctlDemo.btnGetAllSemaphoreValuesClick(Sender: TObject);
var
    SemValues: array[0..0] of integer;
begin
    //Although we only have a binary semaphore, this demonstrates
    //using the GETALL command.
    semctl(SemaphoreID, 0, GETALL, @SemValues);

    mmInfo.Lines.Add('The Value of the semaphore is '+
        inttostr(SemValues[0]));
end;

procedure TfrmSemctlDemo.btnSetPermissionsClick(Sender: TObject);
var
    buf: TSemaphoreIdDescriptor;
    intRes: integer;
begin
    //This changes the permissions for the semaphore,
    //but can only change the uid, gid and mode
    buf.sem_perm.uid := strtoint(edUserID.Text);
    buf.sem_perm.gid := strtoint(edGroupID.Text);
    buf.sem_perm.mode := strtoint(edMode.Text);

```

```

intRes := semctl(SemaphoreID, 0, IPC_SET, @buf);
if intRes <> -1 then
    mmInfo.Lines.Add('The new permissions have been set.')
else
    mmInfo.Lines.Add('The new permissions have not been set.')
end;

```

semget LibC.pas

Syntax

```

function semget(
    __key: key_t;
    __nsems: Integer;
    __semflg: Integer
):Integer;

```

Description

The `semget` function obtains a handle to an existing semaphore or creates a new semaphore and returns the handle of the created semaphore.

Parameters

`__key`: This is a valid IPC key which will be a predefined number, the `IPC_PRIVATE` constant, which is used when creating a new message queue, or a value taken from a call to `ftok`.

`__nsems`: When creating a semaphore, this value contains how many semaphores to create. If the function does not specify to create a semaphore, this value should be left as 0.

`__semflg`: This option specifies the permissions for access control to the semaphore in a similar way that access is controlled to a file. Using the `IPC_CREAT` option in this flag will create the semaphore(s) if they have not been created already. Using the `IPC_EXCL` option in the function will result in the function failing if the semaphore already exists.

Return Value

When successful, the function returns the handle to the semaphore, which will be a positive integer value; otherwise, the function will return `-1` for an error.

See Also

`semctl`, `semop`

Example

See Listing 5.24 - Obtaining a semaphore handle.

semop LibC.pas

Syntax

```

function semop(
    __semid: Integer;

```

```

__sops: PSemaphoreBuffer;
__nsops: size_t;
):Integer;

```

Description

This function performs a number of semaphore operations on an active semaphore. The semaphore operation that is performed is normally a change in the value of the semaphore.

Parameters

__semid: This is the handle to a semaphore that was returned by a successful call to `semget`.

__sops: This parameter points to a buffer of semaphore operations to perform. The operations are contained within a `TSemaphoreBuffer` record. Within the `TSemaphoreBuffer` record, `sem_num` represents the semaphore number to change. This value will typically be 0 for a binary semaphore, but if you created multiple semaphores, then this option can be a valid element number. The `sem_op` field represents the change in the value of the semaphore, which is a negative value if you are taking a semaphore resource value or a positive value when you are increasing the value of the semaphore. The `sem_flg` field allows you to include options for the semaphore operation such as `IPC_NOWAIT`, which will result in the function returning an error if the function will not return immediately. `SEM_UNDO` can also be used in the `sem_flg` field to undo the operation that is being performed when the process is finished.

```

type
  sembuf = record
    sem_num: Smallint;      { semaphore number }
    sem_op: Smallint;       { semaphore operation }
    sem_flg: Smallint;      { operation flag }
  end;

```

```

TSemaphoreBuffer = sembuf;
PSemaphoreBuffer = ^TSemaphoreBuffer;

```

__nsops: This is the number of `TSemaphoreBuffer` operations that are located at the address in the `__sops` parameter.

Return Value

The function returns 0 when successful and -1 when unsuccessful.

See Also

`semget`, `semctl`

Example

Listing 5.45 - Using the `semop` function

```

procedure TfrmSemaphoreDemo.CreateSemaphore;
var
  Operations: array[0..1] of TSemaphoreBuffer;
begin

```

```

//Create the semaphore
FSemaphoreID := semget(1222, 1, DEFFILEMODE or IPC_CREAT);

//Initialize the Semaphore
if semctl(FSemaphoreID, 0, SETVAL, 5) = -1 then
begin
  Showmessage('The semaphore could not be initialised.');
```

Exit;

```
end;

//Perform Semaphore Operations
with Operations[0] do
begin
  sem_num := 0;
  sem_op := -1;
  sem_flg := SEM_UNDO;
end;
with Operations[1] do
begin
  sem_num := 0;
  sem_op := 1;
  sem_flg := SEM_UNDO;
end;

if semop(FSemaphoreID, @Operations, 2) = -1 then
  Showmessage('The semaphore operations could not be completed.');
```

end;

shmat LibC.pas

Syntax

```

function shmat(
  __shmid: Integer;
  __shmaddr: Pointer;
  __shmflg: Integer
):Pointer;
```

Description

The shmat function will attach shared memory held by the Linux kernel to the memory space of the current process, so that the local process can access the shared memory transparently.

Parameters

__shmid: This is the shared memory ID that was returned to a previous call to shmget.

__shmaddr: When used in conjunction with the SHM_RND flag in the __shmflg parameter, this specifies the memory address of where the shared memory will be mapped to. If this parameter is set to nil, the location of where the shared memory will be mapped will be handled by the kernel.

__shmflg: This parameter contains the options for the shared memory resource. This is the permissions mask of which user can connect to the shared memory. If the flag includes the

SHM_RND option, the value in `_shmaddr` will be used to calculate when the shared memory will be accessible from within the local process. If the SHM_RDONLY option is included, the shared memory will be open for read-only access.

Return Value

If successful, the function will return a valid pointer to where the shared memory is accessible from within the local process. If the function was unsuccessful, the function returns nil.

See Also

`shmctl`, `shmdt`, `shmget`

Example

See Listing 5.34 - The cafe reservation application.

shmctl LibC.pas

Syntax

```
function shmctl(
  __shmid: Integer;
  __cmd: Integer;
  __buf: PSharedMemIdDescriptor;
):Integer;
```

Description

The `shmctl` function performs a specific command on a shared memory resource allocated by the `shmget` function. The available commands for this function are to remove the shared memory resource, obtain the properties of the shared memory, or set new security permissions for the shared memory resource.

Parameters

`__shmid`: This is the shared memory handle that was returned by a successful call to `shmget`.

`__cmd`: This is the command that is to be executed. This can be one of three values. `IPC_STAT` retrieves the details of the shared memory into the records that the `_buf` parameter points to. `IPC_SET` assigns the permission values of the `TSharedMemIdDescriptor`'s `shm_perm` property to the shared memory resource. When the permissions are assigned only the uid, gid, and mode properties will actually be changed in the shared memory resource. The last command is the `IPC_RMID` command, which will remove the shared memory from the system when all processes have detached from the shared memory.

`__buf`: This is a pointer to a `TSharedMemIdDescriptor` record to read or set permissions when using the `IPC_STAT` or `IPC_SET` functions. When using the `IPC_RMID` option, this parameter is ignored and should be set to nil.

Return Value

The function returns 0 when successful and -1 when unsuccessful.

See Also

shmat, shmdt, shmget

Listing 5.46 - Using the TSharedMemIdDescriptor record

```

type
  ipc_perm = {packed} record
    __key: __key_t;           { Key. }
    uid: __uid_t;             { Owner's user ID. }
    gid: __gid_t;             { Owner's group ID. }
    cuid: __uid_t;            { Creator's user ID. }
    cgid: __gid_t;            { Creator's group ID. }
    mode: Smallint;           { Read/write permission. }
    __pad1: Smallint;
    __seq: Smallint;          { Sequence number. }
    __pad2: Smallint;
    __unused1: LongWord;
    __unused2: LongWord;
  end;
  {$EXTERNALSYM ipc_perm}
  TIpcPermission = ipc_perm;
  PipcPermission = ^TIpcPermission;

  shmid_ds = record
    shm_perm: ipc_perm;       { operation permission struct }
    shm_segsz: size_t;         { size of segment in bytes }
    shm_atime: __time_t;       { time of last shmat() }
    __unused1: LongWord;
    shm_dtime: __time_t;       { time of last shmdt() }
    __unused2: LongWord;
    shm_ctime: __time_t;       { time of last change by shmctl() }
    __unused3: LongWord;
    shm_cpid: __pid_t;         { pid of creator }
    shm_lpid: __pid_t;         { pid of last shmop }
    shm_nattch: shmatt_t;      { number of current attaches }
    __unused4: LongWord;
    __unused5: LongWord;
  end;

  TSharedMemIdDescriptor = shmid_ds;
  PSharedMemIdDescriptor = ^TSharedMemIdDescriptor;

```

Example

See Listing 5.33 - Using the shmctl function.

shmdt LibC.pas**Syntax**

```
function shmdt(
  __shmaddr: Pointer
):Integer;
```

Description

The shmdt function will detach shared memory mapped in to process space that was previously mapped by a call to shmat.

Parameters

`__shmaddr`: This is the pointer returned from a previous call to shmat that specifies the location of where the shared memory is mapped to the local process.

Return Value

If the function is successful, it returns 0. If the function was unsuccessful, it returns -1. The most likely errors as a result of using this function are that either the memory address in the pointer is invalid or the shared memory has already been detached or has never been attached.

See Also

shmat, shmget, shmctl

Example

See Listing 5.34 - The cafe reservation application.

shmget LibC.pas**Syntax**

```
function shmget(
  __key: key_t;
  __size: size_t;
  __shmflg: Integer
):Integer;
```

Description

This function will either return a handle to a block of shared memory that has been allocated or create a new block and return the handle of the created shared memory.

Parameters

`__key`: This is a valid IPC key to create/open a previously created shared memory resource or the constant `IPC_PRIVATE` to create a uniquely private key for this process.

`__size`: This is the size of the memory block to return. If the `IPC_CREAT` option is used in the `shmflg` parameter, this parameter represents the size of the memory block that will be allocated.

`__shmflg`: This parameter specifies the permission level of this shared memory resource. The permission level for the shared memory resource uses the same principles of a file resource shown in Table 3.2. This parameter may also include the option `IPC_CREAT`, which specifies that the shared memory should be created. If the option `IPC_EXCL` is included, the function will fail if the shared memory was trying to be created but already exists.

Return Value

Upon success, the function returns a valid handle to a shared memory block. This value should be used when calling any of the shared memory functions such as the `shmat`, `shmdt`, and `shmctl` functions. If the function was unsuccessful, it returns `-1`.

See Also

`shmat`, `shmdt`, `shmctl`

Example

See Listing 5.31 - Creating a shared memory handle.

sigaction *LibC.pas*

Syntax

```
function sigaction(
  SigNum: Integer;
  Action: PSigAction;
  OldAction: PSigAction;
):Integer;
```

Description

The `sigaction` function tells the process which `TSigAction` function to call when a particular signal is received by the process. It also allows the retrieval of the current signal handler so that signal management can be restored such as when different signal handlers are being loaded from a shared object.

Parameters

`SigNum`: This is the signal number to use such as a value from Listing 5.1.

`Action`: This is a pointer to a `TSigAction` procedure. A `TSigAction` is a procedural type that takes a single integer parameter and uses the `cdecl` calling convention.

`OldAction`: When this function returns, this parameter returns the value of the previous `TSigAction` used for the signal in the `SigNum` parameter. This is normally used to allow rollback of the signal handlers.

Return Value

If the function was successful, it returns 0. Otherwise, the function returns -1 when it fails.

See Also

signal

Example

See Listing 5.7 - Using signal sets.

sigaddset LibC.pas**Syntax**

```
function sigaddset(
  var SigSet: TSigSet;
  SigNum: Integer;
):Integer;
```

Description

The sigaddset function adds a signal denoted by SigNum to the signal set SigSet.

Parameters

SigSet: This is the signal set which will not include the added signal.

SigNum: This is the signal that will be part of the signal set.

Return Value

The function returns 0 when successful and -1 if it fails.

See Also

sigdelset, sigemptyset, sigfillset

Example**Listing 5.47 - Using the sigaddset function**

```
var
  MySignalSet: TSigSet;
begin
  //Set up the first signal set
  sigemptyset(MySignalSet);
  sigaddset(MySignalSet, SIGPIPE);
end;
```

sigandset LibC.pas***Syntax***

```
function sigandset(
  var SigSet: TSigSet;
  const Left: TSigSet;
  const Right: TSigSet;
):Integer;
```

Description

This function calculates the intersection of two signal sets and returns the resulting intersection signal set.

Parameters

SigSet: This is where the result of the intersection calculation is returned. This value needs to be a valid TSigSet variable.

Left: This parameter is a valid signal set. After the function call, only signals that are in both the Left and the Right parameters will be stored in the SigSet parameter.

Right: This is also a valid signal set. Like the Left parameter, after the function is called, only signals that are in both the Left and the Right parameters will be stored in the SigSet parameter.

Return Value

When successful, the function returns 0. If the function could not be completed, the function returns -1.

See Also

sigorset

Example**Listing 5.48 - Using the sigandset function**

```
var
  MySignalSet: TSigSet;
  SigSet1, SigSet2: TSigSet;
begin
  //Set up the first signal set
  sigemptyset(SigSet1);
  sigaddset(SigSet1, SIGPIPE);
  sigaddset(SigSet1, SIGALRM);
  sigaddset(SigSet1, SIGINT);

  //Set up the second signal set
  sigemptyset(SigSet2);
  sigaddset(SigSet2, SIGPIPE);
  sigaddset(SigSet2, SIGCHLD);

  //Let's get the union of the two sets
```

```

sigandset(MySignalSet, SigSet1, SigSet2);

if sigismember(MySignalSet, SIGALRM) = 1 then
    showmessage('The signal set includes SIGALRM.')
else
    showmessage('The signal set does not include SIGALRM.');
```

sigdelset ***LibC.pas***

Syntax

```

function sigdelset(
    var SigSet: TSigSet;
    SigNum: Integer;
):Integer;
```

Description

This function removes a signal element from a signal set.

Parameters

SigSet: This is the signal set from which the signal will be removed.

SigNum: This is the signal to remove from the signal set and should contain a valid signal identifier.

Return Value

When successful, this function returns 0. If the function failed, it returns -1.

See Also

sigaddset, sigemptyset, sigfillset

Example

Listing 5.49 - Using the sigdelset function

```

var
    MySignalSet: TSigSet;
begin
    sigfillset(MySignalSet);
    sigdelset(MySignalSet, SIGALRM);
end;
```

sigemptyset ***LibC.pas***

Syntax

```

function sigemptyset(
    var SigSet: TSigSet;
):Integer;
```

Description

This function removes all signal entries from a signal set. This function is normally called when initializing a signal mask with only a few entries.

Parameters

SigSet: This is a valid signal set that will be cleared of all its signal elements.

Return Value

This function returns 0 when successful and -1 when it fails.

See Also

sigfillset

Example**Listing 5.50 - Using the sigemptyset function**

```
var
  MySignalSet: TSigSet;

procedure SetupSignalSet;
begin
  //Set up the signal set to only contain SIGALRM
  sigemptyset(MySignalSet);
  sigaddset(MySignalSet, SIGALRM);
end;
```

sigfillset LibC.pas**Syntax**

```
function sigfillset(
  var SigSet: TSigSet;
):Integer;
```

Description

This function fills a signal set with all signal values. This function is used to initialize a signal set to all the values, which will often be used as a signal mask.

Parameters

SigSet: This is a valid signal set that will include all signals after this function call.

Return Value

This function returns 0 when successful and -1 when it fails.

See Also

sigemptyset

*Example***Listing 5.51 - Using the sigfillset function**

```

var
  MySignalSet: TSigSet;
begin
  //initialize the signal set to contain all the signals
  sigfillset(MySignalSet);
  sigdelset(MySignalSet, SIGALRM);
end;

```

siggetmask* LibC.pasSyntax*

```
function siggetmask:Integer;
```

Description

The siggetmask function returns the process's current signal mask that was applied to the process using the sigsetmask function.

Return Value

The function always returns the signal mask for the current process.

See Also

sigsetmask

sighold* LibC.pasSyntax*

```

function sighold(
  SigNum: Integer;
):Integer;

```

Description

This function will add the signal passed into the SigNum parameter to be blocked by the current process. It is the equivalent of using sigprocmask to block a signal set that contains only the signal SigNum.

Parameters

SigNum: This is the signal to block, such as SIGCHLD.

Return Value

The function returns 0 when successful and -1 when it fails.

See Also

sigprocmask, sigrelse

*Example***Listing 5.52 - Using the sighold function**

```

procedure TfrmBlockingSignalsTestForm.btnBlockTheSignalsClick(Sender: TObject);
begin
    //Using the sighold function we can obtain the same results as
    //using the SIG_BLOCK mode with the sigprocmask function
    sighold(SIGUSR1);
    sighold(SIGUSR2);
end;

```

sigignore LibC.pas*Syntax*

```

function sigignore(
    SigNum: Integer;
):Integer;

```

Description

By using the sigignore function, the application is told to ignore the signal passed into the SigNum parameter. This is the equivalent of calling signal(SigNum, SIG_IGN).

Parameters

SigNum: This is the signal to ignore.

Return Value

The function returns 0 when successful and -1 when it fails.

See Also

signal, sigaction

*Example***Listing 5.53 - Using the sigignore function**

```

procedure TfrmBlockingSignalsTestForm.btnIgnoreAlarmsClick(
    Sender: TObject);
begin
    //Ignore the signal SIGALRM. This call is the
    //equivalent of signal(SIGUSR1, SIG_IGN);
    sigignore(SIGUSR1);
end;

```

sigemptyset LibC.pas*Syntax*

```

function sigemptyset(
    const SigSet: TSigSet
):Integer;

```

Description

The `sigemptyset` function checks to see if a signal set contains no signals.

Parameters

`SigSet`: This is a valid signal set that will be examined.

Return Value

If the signal set has any signal elements, the function returns 0. If the signal set is empty, the function returns 1.

See Also

`sigemptyset`

*Example***Listing 5.54 - Using the `sigemptyset` function**

```
var
  MySignalSet: TSigSet;
begin
  //initialize the signal set to contain all the signals
  sigemptyset(MySignalSet);
  if sigemptyset(MySignalSet) = 1 then
    showmessage('The Signal Set is empty.')
  else
    showmessage('The Signal Set has elements.')
end;
```

sigismember* LibC.pasSyntax*

```
function sigismember(
  const SigSet: TSigSet;
  SigNum: Integer;
):Integer;
```

Description

The `sigismember` function determines if a particular signal is included in the values of a signal set.

Parameters

`SigSet`: This is the signal set that is being examined.

`SigNum`: This is the signal value for which to look.

Return Value

This function returns 1 if the signal is found in the signal set and 0 if the signal is not in the signal set.

See Also

sigisemptyset

*Example***Listing 5.55 - Using the sigismember function**

```

var
  MySignalSet: TSigSet;
begin
  //initialize the signal set to contain all the signals
  sigfillset(MySignalSet);
  if sigismember(MySignalSet, SIGPIPE) = 1 then
    showmessage('The signal set includes SIGPIPE.')
  else
    showmessage('The signal set does not include SIGPIPE.');
```

end;

signal LibC.pas*Syntax*

```

function signal(
  SigNum: Integer;
  Handler: TSignalHandler;
):TSignalHandler;
```

Description

The signal function sets up a handler for the SigNum signal, so that the Handler function will be executed whenever the signal is received by the current process. Using the signal function is the older method of handling signals within the application. This is due to the fact that once a signal handler is called, the signal will automatically reinstate the default handler for the action after a call.

Parameters

SigNum: This is the signal that is to be handled and is normally a value from Listing 5.1. If this value is not a valid signal identifier, then the function will most likely fail.

Handler: This is a cdecl procedure type that accepts a single integer which identifies the signal that the process has received. As well as passing in procedure types to this function, you can also pass in some special constants typecast as TSignalHandlers to restore the process's default behavior of a signal or ignore the signal. To restore the default behavior of a signal would require this parameter to receive a TSignalHandler(DIG_DFL) or to ignore the signal the application would pass TSignalHandler(DIG_IGN) to this parameter.

Return Value

This function returns 0 when successful and -1 when it fails.

See Also

kill, alarm

Example

See Listing 5.37 - Using the alarm function.

sigorset *LibC.pas**Syntax*

```
function sigorset(
  var SigSet: TSigSet;
  const Left: TSigSet;

  const Right: TSigSet;
):Integer;
```

Description

This function takes a union of two signal sets to combine and give a new signal set.

Parameters

SigSet: This is where the result of the union of the signal sets will be placed.

Left: This is one of the signal sets that will be used when the union is taken to create the new signal set.

Right: This is also a signal set that will be used when the union is taken to create the new signal set.

Return Value

If the function is successful, it returns 0; otherwise, the function returns -1.

See Also

sigandset

*Example***Listing 5.56 - Using the sigorset function**

```
var
  MySignalSet: TSigSet;
  SigSet1, SigSet2: TSigSet;
begin
  //Set up the first signal set
  sigemptyset(SigSet1);
  sigaddset(SigSet1, SIGPIPE);
  sigaddset(SigSet1, SIGALRM);
  sigaddset(SigSet1, SIGINT);

  //Set up the second signal set
  sigemptyset(SigSet2);
  sigaddset(SigSet2, SIGPIPE);
  sigaddset(SigSet2, SIGCHLD);
```

```
//Let's get the union of the two sets
sigorset(MySignalSet, SigSet1, SigSet2);

if sigismember(MySignalSet, SIGPIPE) = 1 then
  showmessage('The signal set includes SIGPIPE.')
else
  showmessage('The signal set does not include SIGPIPE.');
```

End;

__sigpause LibC.pas

Syntax

```
function __sigpause(
  sig_or_mask: Integer;
  is_sig: Integer
):Integer;
```

Description

The `__sigpause` function will cause a process to wait until either a specific signal or any signal from an integer-based signal mask is obtained by the process. Using the integer-based signal mask is not recommended; it is better to use a function such as `sigwait`.

Parameters

`sig_or_mask`: This parameter is a valid signal such as `SIGALRM` or an integer-based signal mask.

`is_sig`: This parameter determines if the value in `sig_or_mask` is a signal value or an integer signal mask. If this value is non-zero, then the function will wait for the signal passed to `sig_or_mask`; otherwise the function will wait for any signal in the signal mask determined by `sig_or_mask`.

Return Value

The `__sigpause` function is not implemented under Linux and this function will always return `-1`. A call to the `errno` function will return the `ENOSYS` error code.

See Also

`sigwait`, `pause`

Example

Listing 5.57 - Using the `__sigpause` function

```
procedure TfrmBlockingSignalsTestForm.btnWaitForAnAlarmClick(Sender: TObject);
begin
  //Let's wait for a particular signal to arrive.
  __sigpause(SIGALRM, 1);
end;
```

sigpending LibC.pas***Syntax***

```
function sigpending(
  var SigSet: TSigSet;
):Integer;
```

Description

This function will return a list of signals that have been sent to the current process that have been blocked using a function such as `sigprocmask` and are waiting to be unblocked so that the process can handle these signals.

Parameters

`SigSet`: This is a valid `TSigSet` that will contain the signals that are currently pending.

Return Value

The function returns 0 when successful and -1 when unsuccessful.

See Also

`sigprocmask`, `sighold`, `sigrelse`

Example**Listing 5.58 - Using the `sigpending` function**

```
procedure TfrmBlockingSignalsTestForm.btnViewPendingSignalsClick(Sender: TObject);
var
  pendingSignals: TSigSet;
  counter: integer;
begin
  sigpending(pendingSignals);

  //Display the details about the signals that are pending.
  mmPendingSignals.Lines.Clear;
  for counter := 0 to _NSIG - 1 do
    if sigismember(pendingSignals, counter) = 1 then
      mmPendingSignals.Lines.Add('Pending Signal - '+
        StrPas(strsignal(counter)));
  end;
```

sigprocmask LibC.pas***Syntax***

```
function sigprocmask(
  How: Integer;
  SigSet: PSigSet;
  OldSigSet: PSigSet;
):Integer;
```

Description

The `sigprocmask` function defines which signals can be blocked and which signals can be unblocked. It allows the adding and removal of signals from the current set of signals that are being blocked by the process. It also allows you to define exactly which signals can be blocked by the process. The `sigprocmask` function also returns the previous set of the signals that are blocked, so it is also an efficient way to obtain the current list of signals that are blocked by the process.

Parameters

How: This is the operation to perform and will be `SIG_BLOCK`, `SIG_UNBLOCK`, or `SIG_SET`. If the value is `SIG_BLOCK`, then the signals contained in the parameter `SigSet` will be added to the list of signals that are blocked by the process. When this value is `SIG_UNBLOCK`, the signals in the `SigSet` parameter will be removed from the list of signals that are blocked by the process, even if the signals in `SigSet` are not currently blocked. Using the `SIG_SET` value for this parameter will make the signals contained in `SigSet` be blocked and the signals not contained in `SigSet` to not be blocked by the process.

SigSet: This is a pointer to a valid `TSigSet` signal set that is used to perform the specific blocking operation that is specified in the **How** parameter.

OldSigSet: This is a pointer to a `TSigSet` signal set. If this value is not `nil`, the signal set that this parameter points to will contain the list of the signals that were blocked before the call to this function. In this way you can restore a previous list of blocked signals or use this in conjunction with an empty signal set and the `SIG_BLOCK` operation to return the current set of blocked signals.

Return Value

The function returns 0 when successful and -1 when it fails.

See Also

`sighold`, `sigpending`, `sigrelse`

Example

Listing 5.59 - Using the `sigprocmask` function

```
procedure TfrmBlockingSignalsTestForm.btnBlockTheSignalsClick(Sender: TObject);
var
    SignalsToBlock: TSigSet;
    OldSignals: TSigSet;
begin
    //Set up the signal handling
    sigemptyset(SignalsToBlock);

    //Add the custom signals to the signal set
    sigaddset(SignalsToBlock, SIGUSR1);
    sigaddset(SignalsToBlock, SIGUSR2);
```



```

    //Block the current messages
    sigprocmask(SIG_BLOCK, @SignalsToBlock, @OldSignals);
end;

procedure TfrmBlockingSignalsTestForm.btnUnblockTheSIGUSRsClick(
    Sender: TObject);
var
    SignalsToBlock: TSigSet;
    OldSignals: TSigSet;
begin
    //Set up the signal handling
    sigemptyset(SignalsToBlock);

    //Add the custom signals to the signal set
    sigaddset(SignalsToBlock, SIGUSR1);
    sigaddset(SignalsToBlock, SIGUSR2);

    //Block the current messages
    sigprocmask(SIG_UNBLOCK, @SignalsToBlock, @OldSignals);
end;

```

sigrelse LibC.pas

Syntax

```

function sigrelse(
    SigNum: Integer;
):Integer;

```

Description

The `sigrelse` function causes the signal passed into the `SigNum` parameter to not be blocked by the process, regardless if the signal was blocked or not. Calling this function is the equivalent of calling `sigprocmask` using the `SIG_UNBLOCK` command with a signal set that contains only the signal passed into `sigrelse`. Calling this function will cause the signal to be retrieved by the process if that signal is currently pending.

Parameters

`SigNum`: This is the signal that will be allowed to be received by the process.

Return Value

The function returns 0 when successful and -1 when it fails.

See Also

`sigprocmask`, `sigpending`, `sighold`

Example

Listing 5.60 - Using the `sigrelse` function

```

procedure TfrmBlockingSignalsTestForm.btnUnblockTheSIGUSRsClick(
    Sender: TObject);
begin
    //Using the sigrelse function we can obtain the same results as

```

```
//using the SIG_UNBLOCK mode with the sigprocmask function
sigrelse(SIGUSR1);
sigrelse(SIGUSR2);
end;
```

sigset *LibC.pas*

Syntax

```
function sigset(
  SigNum: Integer;
  Disp: TSignalHandler
):TSignalHandler;
```

Description

The sigset function defines which TSignalHandler function to call when a particular signal is retrieved by the process. The sigset function is a simpler way of calling the sigaction function.

Parameters

SigNum: This is the signal that is to have its handler for the process defined.

Disp: This is the TSignalHandler for the particular signal.

Return Value

The function returns the signal handler for the signal before the function call so that it may be restored later.

See Also

signal, sigaction

Example

Listing 5.61 - Using the sigset function

```
procedure DisplayAlarmTime(Signal: integer); cdecl;
begin
  frmSigSetDemo.lcdAlarmTime.Value :=
    TimeToStr(Time);
end;

procedure TfrmSigSetDemo.FormCreate(Sender: TObject);
begin
  //Set up the signal handler for the alarm.
  sigset(SIGALRM, DisplayAlarmTime);
end;

procedure TfrmSigSetDemo.btnSetAlarmClick(Sender: TObject);
begin
  //Send the alarm
  alarm(seSeconds.Value);
end;
```

sigsuspend LibC.pas***Syntax***

```
function sigsuspend(
  const SigSet: TSigset;
):Integer;
```

Description

The sigsuspend function changes the signal mask of the calling process to the signals contained in the SigSet parameter and then waits for a valid signal to arrive. Once the signal has arrived the signal mask is restored to what it was before the function call.

Parameters

SigSet: This is the signal set that will be used as the signal mask temporarily.

Return Value

When successful, this function returns 0; otherwise, it returns -1.

See Also

sigprocmask

Example**Listing 5.62 - Using the sigsuspend function**

```
procedure TfrmBlockingSignalsTestForm.Button1Click(Sender: TObject);
var
  tempSignalMask: TSigSet;
  FullSignalSet: TSigSet;
  SIGUSRSignalAction: TSigAction;
  OldAction: TSigAction;
begin
  sigfillset(tempSignalMask);
  sigdelset(tempSignalMask, SIGUSR1);
  sigdelset(tempSignalMask, SIGUSR2);

  //Temporarily change the blocked signals for the process
  //to the ones in our mask and wait for a valid signal to arrive.
  //We will set signal handlers before we do this however
  sigfillset(FullSignalSet);
  SIGUSRSignalAction.__sigaction_handler := HandleCustomSignals;
  SIGUSRSignalAction.sa_mask := FullSignalSet;
  SIGUSRSignalAction.sa_flags := 0;

  sigaction(SIGUSR1, @SIGUSRSignalAction, @OldAction);

  //Now let's fire up the Signal Handlers
  sigsuspend(tempSignalMask);
end;
```

sigwait LibC.pas***Syntax***

```
function sigwait(
  const SigSet: TSigSet;
  SigNum: PInteger;
):Integer;
```

Description

The sigwait function will wait until a signal contained in SigSet is retrieved. When one of the signals from the signal set is returned, the resulting value is placed in the SigNum parameter.

Parameters

SigSet: This is a valid signal set containing the signals for which the process will wait.

SigNum: This is a pointer to an integer variable that will contain the signal in the signal set.

Return Value

If the function was successful, the function returns 0; otherwise, the function returns -1.

See Also

sigtimedwait, pause

Example**Listing 5.63 - Using the sigwait function**

```
procedure TfrmTheWaitingDemo.btnSendAlarmAndWaitClick(Sender: TObject);
var
  SigAlarmSigSet: TSigSet;
  SignalReturned: integer;
begin
  //Now let's wait for the Alarm signal (SIGALRM)
  sigemptyset(SigAlarmSigSet);
  sigaddset(SigAlarmSigSet, SIGALRM);

  //Send the alarm message (SIGALRM)
  alarm(seTime.Value);

  //Now let's not return until the signal is retrieved
  //The application will look like it has frozen.
  sigwait(SigAlarmSigSet, @SignalReturned);

  Caption := 'Signal Returned was '+StrPas(strsignal(SignalReturned));
end;
```

strsignal LibC.pas**Syntax**

```
function strsignal(
  Sig: Integer
): PChar;
```

Description

The `strsignal` function will return text describing what the signal passed as the `Sig` represents.

Parameters

`Sig`: This parameter is a valid signal number from Table 5.1.

Return Value

When successful, the function returns a null-terminated string containing a description of the signal passed into the `Sig` parameter. If the function was unsuccessful, such as when an integer value that does not represent a signal is passed into the `Sig` parameter, the function returns `nil`.

Example

See Listing 5.10 - Using the `sigpending` function.

sigtimedwait LibC.pas**Syntax**

```
function sigtimedwait(
  const SigSet: TSigSet;
  SigInfo: PSigInfo;
  Timeout: PTimeSpec
): Integer; cdecl;
```

Description

The `sigtimedwait` function waits for any signal in the signal set `SigSet` to be received by the calling process. The main difference between this function and the `sigwait` function is that the `sigtimedwait` function can specify a maximum period of time that the process will wait for a signal. If a signal has been received, the function will place the details about the signal in the `SigInfo` parameter.

Parameters

`SigSet`: This is a valid signal set containing the signals for which the process will wait.

`SigInfo`: This is a pointer to a `TSigInfo` record that will contain the information about the received signal.

```
type
  siginfo = record
    si_signo: Integer;           // Signal number
```

```

    si_errno: Integer;           // Error Code
    si_code: Integer;           // Signal code.
    case Integer of
        0: (_pad: _si_pad);
        1: (_kill: _si_kill);
        2: (_timer: _si_timer);
        3: (_rt: _si_rt);
        4: (_sigchld: _si_sigchld);
        5: (_sigfault: _si_sigfault);
        6: (_sigpoll: _si_sigpoll);
    end;
    siginfo_t = siginfo;
    TSigInfo = siginfo;
    PSigInfo = TSigInfo;

```

Timeout: The Timeout parameter is a pointer to a TTimeSpec record that specifies the number of seconds and nanoseconds the process should wait.

```

type
    timespec = record
        tv_sec: Longint;         // Seconds
        tv_nsec: Longint;       // Nanoseconds
    end;
    TTimeSpec = TimeSpec;
    PTimeSpec = ^TTimeSpec;

```

Return Value

According to the POSIX specification, the function should return 0 when this function executes successfully. If an error occurred, the function would return a valid error code. As this function is not implemented in the current Glibc release, this function will always return -1 and set the error code to ENOSYS.

See Also

sigwait

A Practical Use — A Message Queue Component

This chapter's practical application is a component that simplifies the process of using message queues to a more Kylix-friendly manner. By wrapping these functions, you will be able to send and receive messages in a simple-to-use manner. The companion CD-ROM also contains a version of the MapWatcher client and server application that make use of the new component to demonstrate the use of the API.

Listing 5.64 - A message queue component

```

unit IPCMessageQueue;

(*
    This is a component that allows the ability to control a
    message queue in a method that is more like Kylix.

    Copyright 2001 Glenn Stephens
    for the Tomes of Kylix - Linux API

```

```

*)

interface

uses
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
    Libc, LibcHelperFunctions;

type
    EIPCMessageQueue = class(Exception);

    TIPCMessageQueue = class(TComponent)
    private
        //Queue value exposed in case API calls are needed
        FQueueID: integer;
        FIPCKey: longint;
        FActive: boolean;
        FCreateQueueOnOpen: boolean;
        FDeleteQueueOnClose: boolean;
        FPrivateQueue: boolean;
        FWaitForMessageToArrive: boolean;
        procedure SetActive(const Value: boolean);
        function GetCreatorGroupID: integer;
        function GetCreatorGroupName: string;
        function GetCreatorUserID: integer;
        function GetCreatorUserName: string;
        function GetGroupID: integer;
        function GetGroupName: string;
        function GetReadWriteMode: integer;
        function GetSequenceNumber: integer;
        function GetUserID: integer;
        function GetUserName: string;
        procedure SetCreateQueueOnOpen(const Value: boolean);
        procedure SetDeleteQueueOnClose(const Value: boolean);
        procedure SetIPCKey(const Value: longint);
        procedure SetPrivateQueue(const Value: boolean);
        procedure RefreshQueueInformation;
        procedure SetWaitForMessageToArrive(const Value: boolean);
        function ShouldWeWaitForMessage: integer;
    public
        QueueInformation: TMsgQueueIdDesc;

        //Methods for Opening and Closing the Queue
        procedure Open;
        procedure Close;

        //The method we used to send a message
        procedure SendMessage(var AMessage; MessageSize: integer);

        //Methods to receive messages from the Queue
        function GetTopPriorityMessage(
            var AMessage; RecordSize: integer): boolean;
        function GetTopOfTheQueue(
            var AMessage; RecordSize: integer): boolean;
        function GetPriorityMessage(APriority: integer;
            var AMessage; RecordSize: integer): boolean;
        function GetNextMessageExceptPriority(APriority: integer;

```

```

var AMessage; RecordSize: integer): boolean;

//Functions for receiving information about the Queue
function MessageCount: integer;
function SizeOfMessageQueue: integer;
function ProcessIDThatSentTheLastMessage: integer;
function ProcessIDThatReceivedTheLastMessage: integer;
function TimeOfTheLastMessageSent: TDateTime;
function TimeOfTheLastMessageReceived: TDateTime;

//Constructor/Destructor methods
constructor Create(Owner: TComponent); override;
destructor Destroy; override;

//Loaded method for when the Queue needs to be created
//on the FormCreate
procedure Loaded; override;

//This function will convert a valid file name to an
//IPC key so that it can be used.
procedure UseIPCHandleFromFile(Filename: string; UniqueProjectValue: integer);
published
    property QueueID: integer read FQueueID;
    property IPCKey: longint read FIPCKey write SetIPCKey;
    property PrivateQueue: boolean read FPrivateQueue write SetPrivateQueue;
    property Active: boolean read FActive write SetActive;
    //Information about the Queue
    property CreatorUserID: integer read GetCreatorUserID;
    property CreatorUserName: string read GetCreatorUserName;
    property CreatorGroupID: integer read GetCreatorGroupID;
    property CreatorGroupName: string read GetCreatorGroupName;
    property UserID: integer read GetUserID;
    property UserName: string read GetUserName;
    property GroupID: integer read GetGroupID;
    property GroupName: string read GetGroupName;
    property ReadWriteMode: integer read GetReadWriteMode;
    property SequenceNumber: integer read GetSequenceNumber;
    property WaitForMessageToArrive: boolean read FWaitForMessageToArrive write
        SetWaitForMessageToArrive;
    property CreateQueueOnOpen: boolean read FCreateQueueOnOpen write
        SetCreateQueueOnOpen;
    property DeleteQueueOnClose: boolean read FDeleteQueueOnClose write
        SetDeleteQueueOnClose;
end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('TomesOfKylis', [TIPCMessageQueue]);
end;

function TIPCMessageQueue.GetCreatorGroupID: integer;
begin
    RefreshQueueInformation;

```



```

    Result := QueueInformation.msg_perm.cgid;
end;

function TIPCMessageQueue.GetCreatorGroupName: string;
begin
    RefreshQueueInformation;
    Result := GroupIDToGroupName(QueueInformation.msg_perm.cgid);
end;

function TIPCMessageQueue.GetCreatorUserID: integer;
begin
    RefreshQueueInformation;
    Result := QueueInformation.msg_perm.cuid;
end;

function TIPCMessageQueue.GetCreatorUserName: string;
begin
    RefreshQueueInformation;
    Result := UserIDToUserName(QueueInformation.msg_perm.cuid);
end;

function TIPCMessageQueue.GetGroupID: integer;
begin
    RefreshQueueInformation;
    Result := QueueInformation.msg_perm.gid;
end;

function TIPCMessageQueue.GetGroupName: string;
begin
    RefreshQueueInformation;
    Result := GroupIDToGroupName(QueueInformation.msg_perm.gid);
end;

function TIPCMessageQueue.GetReadWriteMode: integer;
begin
    RefreshQueueInformation;
    Result := QueueInformation.msg_perm.mode;
end;

function TIPCMessageQueue.GetSequenceNumber: integer;
begin
    RefreshQueueInformation;
    Result := QueueInformation.msg_perm.__seq;
end;

function TIPCMessageQueue.GetUserID: integer;
begin
    RefreshQueueInformation;
    Result := QueueInformation.msg_perm.uid;
end;

function TIPCMessageQueue.GetUserName: string;
begin
    RefreshQueueInformation;
    Result := UserIDToUserName(QueueInformation.msg_perm.uid);
end;

```

```

procedure TIPCMessageQueue.Close;
begin
    //Delete the message Queue.
    if FDeleteQueueOnClose then

        msgctl(FQueueID, IPC_RMID, nil);

        FActive := false;
    end;

constructor TIPCMessageQueue.Create(Owner: TComponent);
begin
    inherited;

    FActive := false;
    FWaitForMessageToArrive := true;
end;

destructor TIPCMessageQueue.Destroy;
begin
    if not Active then
        Close;
    inherited;
end;

function TIPCMessageQueue.GetPriorityMessage(APriority: integer;
    var AMessage; RecordSize: integer): boolean;
begin
    if not Active then
        raise EIPCMessageQueue.Create('You cannot receive from the message '+
            'queue unless the queue is open.');
```

Result := msgrcv(FQueueID, AMessage,
RecordSize - sizeof(Longint), APriority, ShouldWeWaitForMessage) <> -1;

```

end;

function TIPCMessageQueue.GetNextMessageExceptPriority(APriority: integer;
    var AMessage; RecordSize: integer): boolean;
begin
    if not Active then
        raise EIPCMessageQueue.Create('You cannot receive from the message '+
            'queue unless the queue is open.');
```

Result := msgrcv(FQueueID, AMessage,
RecordSize - sizeof(Longint), APriority, MSG_EXCEPT or ShouldWeWaitForMessage)
<> -1;

```

end;

function TIPCMessageQueue.GetTopOfTheQueue(
    var AMessage; RecordSize: integer): boolean;
begin
    if not Active then
        raise EIPCMessageQueue.Create('You cannot receive from the message '+
            'queue unless the queue is open.');
```

Result := msgrcv(FQueueID, AMessage,
RecordSize - sizeof(Longint), 0, ShouldWeWaitForMessage) <> -1;

```

end;

function TIPCMessageQueue.GetTopPriorityMessage(
    var AMessage; RecordSize: integer): boolean;
begin
    if not Active then
        raise EIPCMessageQueue.Create('You cannot receive from the message '+
            'queue unless the queue is open.');
```

Result := msggrcv(FQueueID, AMessage,
RecordSize - sizeof(Longint), -1, ShouldWeWaitForMessage) <> -1;

```

end;

procedure TIPCMessageQueue.Loaded;
begin
    inherited;

    if FActive then
        begin
            //We have to create the Queue with the appropriate information
            Open;
        end;
end;

function TIPCMessageQueue.MessageCount: integer;
begin
    RefreshQueueInformation;
    //Returns the number of messages that are in the queue
    Result := QueueInformation.msg_qnum;
end;

procedure TIPCMessageQueue.Open;
begin
    //Opens the message Queue
    Active := false;

    //We create it if we need to
    if FCreateQueueOnOpen then
        FQueueID := msgget(FIPCKey, 600 or IPC_CREAT)
    else
        FQueueID := msgget(FIPCKey, 600);

    if FQueueID = -1 then
        begin
            //There was a problem creating the message queue
            raise EIPCMessageQueue.Create('The Queue could not be created');
        end;

    RefreshQueueInformation;
    FActive := true;
end;

function TIPCMessageQueue.ProcessIDThatReceivedTheLastMessage: integer;
begin
    RefreshQueueInformation;

```

```

    Result := QueueInformation.msg_lrpId;
end;

function TIPCMessageQueue.ProcessIDThatSentTheLastMessage: integer;
begin
    RefreshQueueInformation;
    Result := QueueInformation.msg_lspId;
end;

procedure TIPCMessageQueue.SetActive(const Value: boolean);
begin
    FActive := Value;
end;

function TIPCMessageQueue.SizeOfMessageQueue: integer;
begin
    RefreshQueueInformation;
    //Returns the size of the Queue in bytes.
    Result := QueueInformation.msg_qbytes;
end;

function TIPCMessageQueue.TimeOfTheLastMessageReceived: TDateTime;
begin
    RefreshQueueInformation;
    Result := IntTimeToDateTime(QueueInformation.msg_rtime);
end;

function TIPCMessageQueue.TimeOfTheLastMessageSent: TDateTime;
begin
    RefreshQueueInformation;
    Result := IntTimeToDateTime(QueueInformation.msg_ctime);
end;

procedure TIPCMessageQueue.SendMessage(var AMessage;
    MessageSize: integer);
var
    intErrorCode: integer;
begin
    if not Active then
        raise EIPCMessageQueue.Create('You cannot send to the message '+'
            'queue unless the queue is open.');
```

//This function sends the message to the message queue.
 //The MessageSize parameter is actually decreased by
 //the size of an Longint. As the first field of the
 //record is an integer, its size is not included in the
 //message that is sent.

```

    if msgsnd(FQueueID, AMessage,
        MessageSize - sizeof(Longint), 0) = -1 then
    begin
        intErrorCode := errno;
        raise EIPCMessageQueue.Create('The Message could not be sent: ' +
            strerror(intErrorCode));
    end;
end;
end;

```

```

procedure TIPCMessagesQueue.SetCreateQueueOnOpen(const Value: boolean);
begin
    FCreateQueueOnOpen := Value;
end;

procedure TIPCMessagesQueue.SetDeleteQueueOnClose(const Value: boolean);
begin
    FDeleteQueueOnClose := Value;
end;

procedure TIPCMessagesQueue.SetIPCKey(const Value: longint);
begin
    FIPCKey := Value;
end;

procedure TIPCMessagesQueue.SetPrivateQueue(const Value: boolean);
begin
    //This sets up a private message queue, which can only be accessed
    //by this process.
    FPrivateQueue := Value;

    if FPrivateQueue then
        FIPCKey := IPC_PRIVATE;
end;

procedure TIPCMessagesQueue.UseIPCHandleFromFile(Filename: string;
    UniqueProjectValue: integer);
var
    Handle: integer;
begin
    //This function obtains a handle to a filename so that it can
    //be used as a message queue so that other processes can access
    //it.
    Handle := ftok(PChar(Filename), UniqueProjectValue);
    if Handle = -1 then
        raise EIPCMessagesQueue.Create('The IPC Handle could not be created.');
```

```

    IPCKey := Handle;
end;

procedure TIPCMessagesQueue.RefreshQueueInformation;
begin
    //Let's get the information about the queue and load the
    //properties of the TIPCMessagesQueueAccessInformation
    msgctl(FQueueID, IPC_STAT, @QueueInformation);
end;

procedure TIPCMessagesQueue.SetWaitForMessageToArrive(const Value: boolean);
begin
    FWaitForMessageToArrive := Value;
end;

function TIPCMessagesQueue.ShouldWeWaitForMessage: integer;
begin
    //This property sets the flags so that the function will automatically
    //return if the message requested is not in the queue.
    if FWaitForMessageToArrive then
        Result := 0
```

```
    else  
        Result := IPC_NOWAIT;  
    end;  
  
end.
```

Conclusion

There are many ways to communicate between processes under Linux. If you have banged your head against the desk trying to speed up data storage and retrieval between a database or files, the Linux API techniques presented in this chapter should give your applications a real speed improvement. On real-time systems, the techniques demonstrated in this chapter are essential for speedy performance.

But interprocess communication does not stop there. This chapter has demonstrated the techniques used for communicating with processes on the same machine. The next logical step is to communicate between processes on different machines. This is exactly the journey we will take in the chapter on sockets — the great communicator.

But first we need to discuss how to perform multiple tasks in an application using threading.

POSIX Threads

Introduction

Have you ever read the newspaper while eating breakfast? Or perhaps you pay bills while watching TV. Doing many things at once enables us to make more efficient use of the time we have.

Your Linux applications also have the ability to perform multiple tasks at once to make efficient use of the application's time. Each task that is being performed is known as a thread. When a process starts, it will have only one thread, which executes the program code for a console application. For a CLX application, the process will wait for user responses and handle them in a single thread.

Having multiple threads in an application often benefits the efficiency of your applications. Imagine having a separate thread load the result of a large database query so the user is not forced to wait until it loads. Or how about a thread that looks for spelling errors while you are still working on a document. These are just some of the benefits of threads, and in this chapter you will learn to make the most of these benefits.

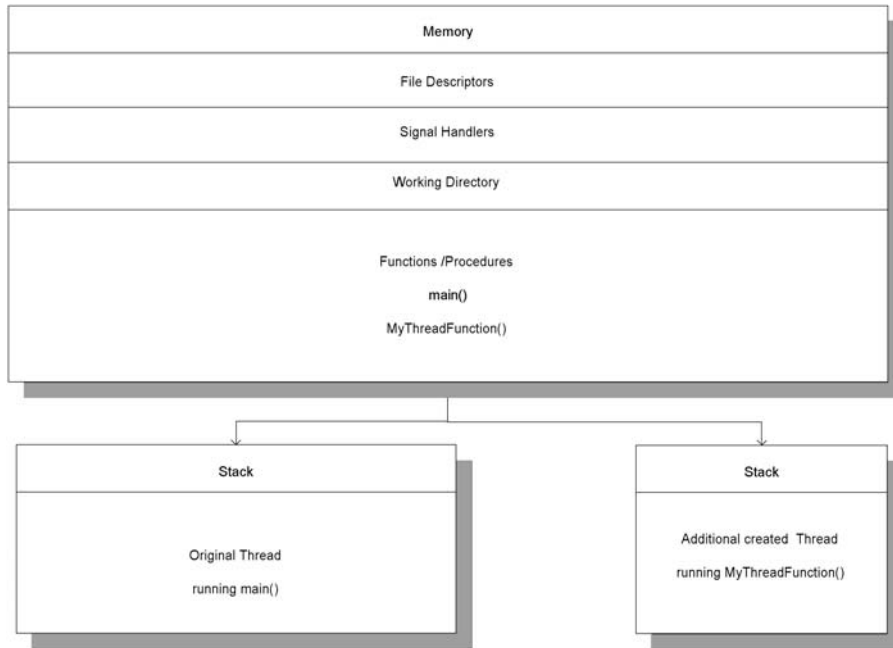
In this chapter we look at POSIX threads, also known as Pthreads, which is how Linux implements threads. You will look at how to create multiple threads, whether or not to use threads or processes, integrating POSIX threads with the TThread object, and the various synchronization methods that can be used in multi-threaded applications.

What are Threads? — An Introduction

So what are threads? A thread is a separate task that runs independently of other tasks within a process. You saw in Chapter 4 that each process has its own memory space, stack space, environment variables, and executable code. When a process is forked, a copy is made of the entire process and the new process executes independently of the original process.

Threads have a different purpose, however. When a process forks, it is normally to pass off handling of a request to another process so that each process can run in isolation. A thread is used to perform the same tasks, but the individual tasks can access the same resources of the process, such as memory and open files as shown in Figure 6.1.

Figure 6.1 - Multi-thread sharing the same resources



LinuxThreads — The POSIX Standard on Linux

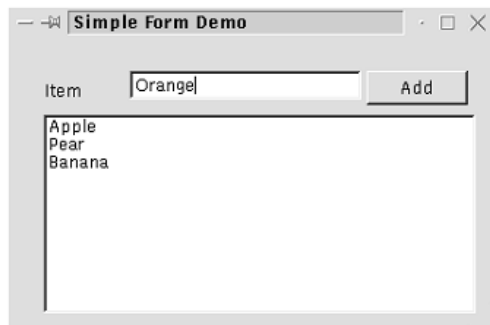
The threading support for Linux is actually a bit of a hack. For developers who come from a Windows background, you may think that creating a thread requires fewer resources than creating a process. The bad news is that threads under Linux are actually implemented by using the clone function. The clone function actually creates a new process that shares some of the details of the original process such as its memory space, signal, and file descriptor information. The clone function is similar to the fork function in that internally within the Linux kernel, it appears that a new process is created. The thread is then scheduled by the Linux kernel, which results in a more efficient allocation of CPU time per thread. Using this method also allows the threads to be executed on different processors in a multiprocessor system.

Threads under Linux are actually implemented using several of the POSIX standards, which include the POSIX 1003.1c API standard through to the POSIX 1003.1j standard. The POSIX 1003.1c standard implements a generic group of functions that are used on different operating systems worldwide and provide support for multi-threaded applications and synchronization methods such as semaphores, mutexes, and others. Work on the POSIX standards is always evolving. The current POSIX standard, 1003j, involves advanced threading functions such as spinlocks, barriers, and reader/writer locks.

Understanding Threads within a Kylix Application

Before you get too far into working with multi-threaded applications, let's examine what happens normally within a Kylix application. Every Kylix application will have at least one thread, and that one thread handles all the processing.

Figure 6.2 -
A simple form



For console applications, the single thread of execution is quite obvious. When you run a Kylix CLX application, there is still only one thread of execution. If you have a form with several components, such as the one shown in Figure 6.2, within your application your thread will be performing one task at a time. This is most likely to be initializing the application to start, waiting for an event to be received, or responding to an event.

As a result, if the Add button were to be clicked in the form displayed in Figure 6.2, the code in the OnClick event for that button would execute, and no further operations could take place until the event had finished.

This is the nature of all visual applications running under Kylix, and unless code is specifically written to make use of additional threads within the application, the application will always use a single thread.

Figure 6.3 -
System
resources on
a Linux PC

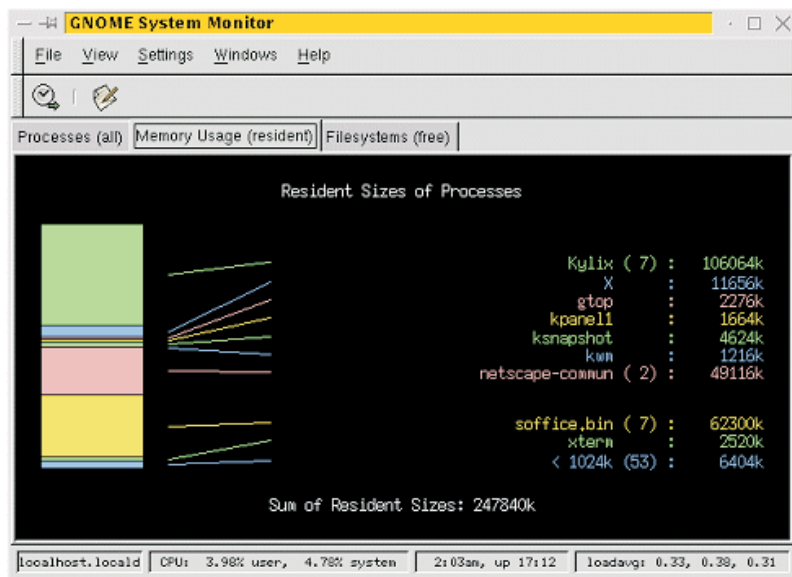


Figure 6.3 shows a view of my system as it was running at that moment, using the GNOME System Monitor utility. If you look at the status bar at the bottom, you will see that the system is actually running at approximately 9 percent of its capacity. This is taking into consideration that I am currently running Kylix, Star Office, Netscape, and many other applications. This should give you the impression that there is usually plenty of room on

your system to run a couple of threads without affecting the performance of the system by much.

Before you dive right in and create a multi-threaded application, however, it is worth determining if your application needs to be multi-threaded at all, or if another technique may be more appropriate, such as splitting the functionality between processes.

Choosing between Threads and Processes

Not every situation will require that your application use threads. Threads add a level of complexity that can cause many development projects to have errors introduced into the applications.

So when should you use threads instead of a process? One instance where you would use threads in an application is when you have the user interface of the application being used by the main thread, and you could have a second thread being used to validate the information being entered by the user without affecting how the user is using the application. This is used in many word-processing applications that may underline an incorrectly spelled word. The spell checking is done by a separate thread.

In the case where two or more tasks need to be executed simultaneously and both tasks need access to the same variables or data, it is often best to use threads. Although the same result may be achieved using shared memory, the extra workload in communicating with the shared memory may not be worth it, and using shared memory has limitations due to the fact that the shared memory's size should be known when the shared memory is initialized.

When another task needs to be done within your application, but the task is independent of the main thread and does not read or write to any of the data that the main thread uses, you may be able to use a thread or a process. Normally the deciding factor is a matter of preference. Take, for example, the simple Web server in Listing 7.14, where when a request is received by the server, the process is forked and the child process that is created handles the request. By forking the request, the tasks of returning content to the HTTP client could go on independently of the main process, which handles the process of accepting connections. Instead of a process being created every time a request comes in, a thread could be created.

There are benefits and drawbacks to using threads in this scenario. On the plus side, multiple threads can be created when the application starts and the threads are then suspended until a request comes through. When a thread comes through, one of the threads can be resumed to handle the individual request. Because no processes need to be created, there will be a performance benefit. On the downside, if the Web server uses threads and the thread causes an error which results in the process crashing, then other requests being handled by other threads will crash also. If the process was forking when it received a request, then if a newly created process that handles the request causes an error, the original process will not be affected. In cases where the application is extremely mission critical and the application should not die, you should probably be using processes instead of threads.

Another question to ask yourself when looking at your application design is if you need threads at all. If you can design your application to perform exactly the same without the need for threads, you should probably do it. The complexity that threading adds to

application development is not worth the effort in many cases. In many software development projects, the ones that are most successful are the ones that have solid, non-complicated architectures. If you can avoid the use of threads in your application, do so.

Error Handling with the POSIX Thread Functions

In Chapter 2, you learned about the standard way that the Linux API functions manage errors, where each function will return 0 if the function is successful or -1 if the function resulted in an error. When the function resulted in an error, a simple call to the `errno` function would return the error code of the function.

The same holds true when you are developing an application with more than one thread. The `errno` function will always return the error code for the last unsuccessful function call on the current thread. Doing this ensures that the executed code in one thread does not affect the error results in another thread.

Reentrant Functions

Each POSIX thread function will return 0 on success and the error code if the function resulted in an error. This is a simple solution for the threaded functions, but there may be functions where an error will affect the result of the `errno` function. It is not just the `errno` function that uses a global variable to store information. The function `readdir`, which reads an entry from a directory, is declared as follows:

```
function readdir(Handle: PDirectoryStream): PDirEnt; cdecl;
```

The function actually returns a pointer to a `TDirEnt` structure that is stored internally. The problem with this function in a multi-threaded application is that if another thread calls this function, then the pointer may point to inconsistent results when dealing with the `PDirEnt` returned by this function.

The Linux API solution to this problem is to create a separate function that can be used in a multi-threaded application by making sure all the information returned. These functions are known as reentrant, meaning that the functions can be entered any time by any thread without inadvertently affecting the outcome of another thread's call to the function. These functions are also known as "thread-safe." If this is applied to the `readdir` API function, the reentrant version of the function is the `readdir_r` function.

```
function readdir_r(Handle: PDirectoryStream; Entry: PDirEnt;
    var __result: PDirEnt): Integer; cdecl;
```

The `readdir_r` function is equivalent in functionality to the `readdir` function, except that it can safely be used in a multi-threaded application. The `_r` which is appended to the function is a convention used within the Linux API that marks the function as a reentrant function. Making effective use of reentrant functions is important to understand before you start creating and developing POSIX thread applications.

Creating Threads

Creating a POSIX thread within your Kylix application using the Linux API is a matter of using the `pthread_create` function.

```
function pthread_create(var ThreadID: TThreadID; Attr: PThreadAttr;
    StartRoutine: TPTHreadFunc; Arg: Pointer): Integer; cdecl;
```

The `pthread_create` function takes several parameters. The first parameter is a `ThreadID` variable, which will store the thread's identifier after the successful function call. The `Attr` parameter defines the attributes for the thread when it is created. Attributes for a POSIX thread are covered later in the chapter, but in many cases you can pass a nil pointer to this parameter for the created thread to begin with default attributes.

The `StartRoutine` parameter is a very important parameter. It defines which procedures should be executed when the thread begins. The `StartRoutine` parameter is a `TPTHreadFunc` parameter, which is a function that takes a single pointer parameter and returns an integer.

```
TPTHreadFunc = function(Parameter: Pointer): Integer; cdecl;
```

When the thread is created, this function will be executing within the context of the newly created thread. When the function within the new thread is finished, the thread has terminated.

Finally, the `Arg` parameter defines the data that is passed to the thread function when it is initially run.

To create a separate thread within your application, you need a function that will execute when the thread is created and some storage for the thread to run. Listing 6.1 demonstrates a simple example of creating a thread which tallies some numbers.

Listing 6.1 - Creating a thread using the `pthread_create` function

```
unit unitMainForm;

interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs, Libc,
    QStdCtrls;

type
    TfrmThreadDemo = class(TForm)
        btnCreateThread: TButton;
        procedure btnCreateThreadClick(Sender: TObject);
    private
        MyThreadID: Cardinal;
    public
        { Public declarations }
    end;

var
    frmThreadDemo: TfrmThreadDemo;

implementation
```

```

{$R *.xfm}

{ Simple Threading example }

var
    Total: integer = 0;

function MyThreadFunction(Data: Pointer): integer; cdecl;
var
    Counter: integer;
begin
    for counter := 0 to 2000 do
        inc(Total, counter);

    Result := 0;
end;

procedure TfrmThreadDemo.btnCreateThreadClick(Sender: TObject);
begin
    if pthread_create(MyThreadID, nil, MyThreadFunction, nil) = 0 then
    begin
        //The Thread could not be created.
        MessageDlg('The thread could not be created.', mtError, [mbOk], 0);
    end;
end;

end.

```

You can see in Listing 6.1 that `MyThreadFunction` is the function that will be executed when the thread begins. The thread executes cleanly and then terminates. When the `pthread_create` function successfully returns, the `MyThreadID` parameter should contain the ID of the thread. The thread ID returned is used to uniquely identify the thread and is used in many other POSIX threading functions.

It is worth noting that a call to the `pthread_create` function, like most thread functions, will return 0 when the function successfully returns and an error code if the function fails.

While this function does create a separate thread, it remains impractical, as the original thread will not know when the thread has finished calculating the total. One of the ways we can know when the thread is finished is to wait for it.

Thread Attributes

In the previous section, when a thread was created using the `pthread_create` function, the second parameter was a pointer to a thread attribute. This parameter is a structure that holds how the thread will operate and allows the thread to be different by default by modifying if the thread is detached or not, how the thread is scheduled, and even the size of the stack for the various threads.

```

type
    pthread_attr_t = record
        __detachstate,
        __schedpolicy: Integer;
        __schedparam: TSchedParam;
        __inheritsched,

```

```

    __scope: Integer;
    __guardsize: TSize_T;
    __stackaddr_set: Integer;
    __stackaddr: Pointer;
    __stacksize: TSize_T;
end;
TThreadAttr = pthread_attr_t;
PThreadAttr = ^TThreadAttr;

```

An attribute is a variable of type `TThreadAttr`, which holds all the information about how the thread will operate. Although this structure holds the information about how the thread will operate, it is not advised to set the fields within this record directly. Instead, there are groups of functions shown in Listing 6.2 designed to set the thread attribute values for you.

Listing 6.2 - Functions to work with thread attributes

```

function pthread_attr_init(var Attr: TThreadAttr): Integer; cdecl;
function pthread_attr_destroy(var Attr: TThreadAttr): Integer; cdecl;

function pthread_attr_setdetachstate(var Attr: TThreadAttr;
    DetachState: Integer): Integer; cdecl;
function pthread_attr_getdetachstate(const Attr: TThreadAttr;
    var DetachState: Integer): Integer; cdecl;
function pthread_attr_setschedparam(var Attr: TThreadAttr;
    Param: PSchedParam): Integer; cdecl;
function pthread_attr_getschedparam(const Attr: TThreadAttr;
    var Param: TSchedParam): Integer; cdecl;
function pthread_attr_setschedpolicy(var Attr: TThreadAttr;
    Policy: Integer): Integer; cdecl;
function pthread_attr_getschedpolicy(const Attr: TThreadAttr;
    var Policy: Integer): Integer; cdecl;
function pthread_attr_setinheritsched(var Attr: TThreadAttr;
    Inherit: Integer): Integer; cdecl;
function pthread_attr_getinheritsched(const Attr: TThreadAttr;
    var Inherit: Integer): Integer; cdecl;
function pthread_attr_setscope(var Attr: TThreadAttr;
    Scope: Integer): Integer; cdecl;
function pthread_attr_getscope(const Attr: TThreadAttr;
    var Scope: Integer): Integer; cdecl;
function pthread_attr_setguardsize(var Attr: TThreadAttr;
    Guardsize: LongWord): Integer; cdecl;
function pthread_attr_getguardsize(const Attr: TThreadAttr;
    var Guardsize: LongWord): Integer; cdecl;
function pthread_attr_setstackaddr(var Attr: TThreadAttr;
    StackAddr: Pointer): Integer; cdecl;
function pthread_attr_getstackaddr(const Attr: TThreadAttr;
    var StackAddr: Pointer): Integer; cdecl;
function pthread_attr_setstack(var Attr: TThreadAttr; StackAddr: Pointer;
    StackSize: size_t): Integer; cdecl;
function pthread_attr_getstack(const Attr: TThreadAttr; var StackAddr: Pointer;
    var StackSize: size_t): Integer; cdecl;
function pthread_attr_setstacksize(var Attr: TThreadAttr;
    StackSize: LongWord): Integer; cdecl;
function pthread_attr_getstacksize(const Attr: TThreadAttr;
    var StackSize: LongWord): Integer; cdecl;

```

The first two functions in Listing 6.2 are `pthread_attr_init` and `pthread_attr_destroy`. These two functions handle the initialization and destruction of the thread attribute. Although you do not have to call these functions, the POSIX standard suggests that you do, as these functions may be implemented differently on different distributions, and with the constant development work done on Linux, the implementation may change later on. The code to create a thread attribute would look similar to the code in Listing 6.3.

Listing 6.3 - Creating a thread attribute

```
procedure CopyFileInBackground(SourceFile, DestFile: string);
var
  FileDetails: PFileToCopy;
  MyThread: ^pthread_t;
  MyThreadAttribute: PThreadAttr;
begin
  //This function will copy a file in a background thread
  new(FileDetails);
  StrPCopy(FileDetails^.SourceFile, SourceFile);
  StrPCopy(FileDetails^.DestinationFile, DestFile);

  //Create the IDs.
  new(MyThread);
  new(MyThreadAttribute);

  //Let's define the attributes for the thread
  pthread_attr_init(MyThreadAttribute^);

  //Because this will run entirely in the background, we will detach the thread
  pthread_attr_setdetachstate(MyThreadAttribute^, PTHREAD_CREATE_DETACHED);

  //Let's create the thread now.
  pthread_create(MyThread^, MyThreadAttribute, BackgroundCopyFile, FileDetails);
end;
```

The other functions in Listing 6.2 are used to set and retrieve particular information from the thread attribute which is telling the thread if it is to be detached from all threads so that no thread waiting for the thread to be finished will be notified, set the scheduling policy and values for the thread, and set and retrieve the size of the stack as well as setting values to prevent the stack from overflowing.

The most common thread attribute you are likely to set is whether or not the thread will be detached from other threads. This means that a call to `pthread_join` is not available to call against this thread. To achieve this, the `pthread_attr_setdetachstate` function is called. The `pthread_attr_setdetachstate` function takes two parameters which consist of the attribute itself and a value stating whether or not the thread is detached. For the last parameter, use the constant `PTHREAD_CREATE_DETACHED` if you want the thread to be unjoinable or the constant `PTHREAD_CREATE_JOINABLE` if you want other threads to be able to join this thread.

Scheduling information is another common task for POSIX threads. The scheduling information about a thread is used to state which threads will receive more of the CPU's time. Setting scheduling attribute information relies on the `pthread_attr_setschedparam`, `pthread_attr_setschedpolicy`, and `pthread_attr_setinheritsched` functions.


```

const
    SCHED_OTHER    = 0;
    SCHED_FIFO     = 1;
    SCHED_RR       = 2;

function pthread_attr_setschedpolicy(var Attr: TThreadAttr;
    Policy: Integer): Integer; cdecl;
function pthread_attr_setschedparam(var Attr: TThreadAttr;
    Param: PSchedParam): Integer; cdecl;
function pthread_attr_setinheritsched(var Attr: TThreadAttr;
    Inherit: Integer): Integer; cdecl;
function pthread_attr_setscope(var Attr: TThreadAttr;
    Scope: Integer): Integer; cdecl;

```

The `pthread_attr_setschedpolicy` function defines how the scheduling is to be calculated. When setting the `Policy` parameter, the available options are `SCHED_RR`, which means that the scheduling will be done using a round robin algorithm; `SCHED_FIFO`, which means that the scheduling will be done on a first-in/first-out basis; and `SCHED_OTHER`, which is where the scheduling priority for the individual thread is defined with the attribute. The `SCHED_FIFO` and `SCHED_RR` options are only available when the threads are running as a superuser, as these are for real-time applications. `SCHED_OTHER` is the default scheduling option for most threads, and when using this option, you should also set the scheduling value used with the `pthread_attr_setschedparam` function.

The `pthread_attr_setschedparam` function defines the thread priority when the thread is running under the `SCHED_FIFO` or `SCHED_RR` policy. When the scheduling policy is `SCHED_OTHER`, the `pthread_attr_setschedparam` function has no effect.

Another option to setting the schedule information is to tell the thread that you can inherit the thread options from the parent of the thread. This is done by passing in the constant `PTHREAD_INHERIT_SCHED` to the `pthread_attr_setinheritsched` function. If you do not want to inherit the scheduling properties of the parent thread, you can use the `PTHREAD_EXPLICIT_SCHED` value in the thread option. When the `PTHREAD_EXPLICIT_SCHED` value is used, the scheduling attributes need to be set with the `pthread_attr_setschedpolicy` and `pthread_attr_setschedparam` functions. The default value for a thread when it is created is to explicitly set the schedule using the `PTHREAD_EXPLICIT_SCHED` option.

The `pthread_attr_setscope` function is used to specify if the scheduling is calculated between all the threads within the system or if the scheduling calculation is done within all the threads in the process. By default, the scheduling is done system wide, due to the fact that Linux's implementation of POSIX threads does not support process-level scheduling. For system-wide scheduling, the constant passed into this function is `PTHREAD_SCOPE_SYSTEM`, which is the only option supported on Linux. The `PTHREAD_SCOPE_PROCESS` option is not available on Linux at all.

The last portion of the thread attribute functions deals with the stack size of the thread. There are several functions available for setting the stack size, location, and protection size. It is unlikely that you will need to alter the values for the stack as the default stack size for a thread is usually sufficient.

Listing 6.4 - Using thread attributes

```

unit unitMainForm;

interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
    QStdCtrls, QComCtrls, Libc, QExtCtrls;

type
    TfrmThreadAttrDemo = class(TForm)
        Label1: TLabel;
        Label2: TLabel;
        cbCreateDetached: TCheckBox;
        Label3: TLabel;
        cbSchedPolicy: TComboBox;
        Label4: TLabel;
        seSchedulingPriority: TSpinEdit;
        cbInheritSchedule: TCheckBox;
        btnCreateCopyThread: TButton;
        Bevel1: TBevel;
        Label5: TLabel;
        edSourceFile: TEdit;
        btnBrowseSourceFile: TButton;
        Label6: TLabel;
        edDestinationFile: TEdit;
        btnBrowseDestinationFile: TButton;
        OpenDialog1: TOpenDialog;
        SaveDialog1: TSaveDialog;
        btnWaitForThread: TButton;
        procedure btnCreateCopyThreadClick(Sender: TObject);
        procedure btnBrowseSourceFileClick(Sender: TObject);
        procedure btnBrowseDestinationFileClick(Sender: TObject);
        procedure btnWaitForThreadClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    frmThreadAttrDemo: TfrmThreadAttrDemo;
    POSIXThread: pthread_t;
    POSIXThreadAttr: pthread_attr_t;
    POSIXSchedParam: TSchedParam;

implementation

{$R *.xfm}

var
    SourceFile, DestinationFile: string;

function CopyAFile(Value: Pointer): integer; cdecl;
var
    Source, Dest: TFileStream;

```

```

begin
    Source := TFileStream.Create(SourceFile, fmOpenRead);
    try
        Dest := TFileStream.Create(DestinationFile, fmCreate);
        try
            Dest.CopyFrom(Source, 0);
        finally
            Dest.Free;
        end;
    finally
        Source.Free;
    end;

    Result := 0;
end;

procedure TfrmThreadAttrDemo.btnCreateCopyThreadClick(Sender: TObject);
begin
    //Now we will create the thread with the specific attribute
    pthread_attr_init(POSIXThreadAttr);

    //Set the detached state
    if cbCreateDetached.Checked then
        pthread_attr_setdetachstate(POSIXThreadAttr, PTHREAD_CREATE_DETACHED)
    else
        pthread_attr_setdetachstate(POSIXThreadAttr, PTHREAD_CREATE_JOINABLE);

    //Set the scheduling
    case cbSchedPolicy.ItemIndex of
        0: pthread_attr_setschedpolicy(POSIXThreadAttr, SCHED_OTHER);
        1: pthread_attr_setschedpolicy(POSIXThreadAttr, SCHED_RR);
        2: pthread_attr_setschedpolicy(POSIXThreadAttr, SCHED_FIFO);
    end;

    //Set the scheduling priority
    POSIXSchedParam.sched_priority := seSchedulingPriority.Value;
    pthread_attr_setschedparam(POSIXThreadAttr, @POSIXSchedParam);

    //Set the inheritable scheduling
    if cbInheritSchedule.Checked then
        pthread_attr_setinheritsched(POSIXThreadAttr, PTHREAD_INHERIT_SCHED)
    else
        pthread_attr_setinheritsched(POSIXThreadAttr, PTHREAD_EXPLICIT_SCHED);

    //Now let's create the thread
    SourceFile := edSourceFile.Text;
    DestinationFile := edDestinationFile.Text;

    pthread_create(POSIXThread, @POSIXThreadAttr, CopyAFile, nil)
end;

procedure TfrmThreadAttrDemo.btnBrowseSourceFileClick(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
        edSourceFile.Text := OpenFileDialog1.FileName;
    end;
end;

```

```

procedure TfrmThreadAttrDemo.btnBrowseDestinationFileClick(Sender: TObject);
begin
    if SaveDialog1.Execute then
        edDestinationFile.Text := SaveDialog1.FileName;
end;

procedure TfrmThreadAttrDemo.btnWaitForThreadClick(Sender: TObject);
begin
    //Wait for the thread,
    if pthread_join(POSIXThread, nil) = EINVAL then
        ShowMessage('The thread is detached.');
```

end;

end.

Waiting for Threads to Finish

When you are working with interprocess communication, there may be times when you will start one or more processes and then wait for the processes to terminate, so that you could analyze the results. The same is true when working with threads. Instead of using the wait function to wait for a process to terminate, we use the `pthread_join` function.

```
function pthread_join(ThreadID: TThreadID; ThreadReturn: PPointer): Integer; cdecl;
```

The `pthread_join` function will result in the calling thread blocking until the thread identified by `pthread_join`'s `ThreadID` parameter has terminated. Once the function returns, the thread has terminated. `ThreadReturn` also contains the termination code for the thread. These are the same termination codes used for processes.

If you were going to wait for the thread that we created to finish, we would change the `TfrmThreadDemo.btnCreateThreadClick` event to look like Listing 6.5.

Listing 6.5 - Waiting for a POSIX thread to finish

```

procedure TfrmThreadDemo.btnCreateThreadClick(Sender: TObject);
begin
    if pthread_create(MyThreadID, nil, MyThreadFunction, nil) = 0 then
        begin
            //Let's wait for the thread to finish
            pthread_join(MyThreadID, nil);
            lblTotal.Caption := inttostr(Total);
        end else
            MessageDlg('The thread could not be created.', mtError, [mbOk], 0);
end;
```

Listing 6.5 demonstrates the calling thread blocking until the thread identified by `MyThreadID` has terminated. In many cases, using the `pthread_join` function is useful, especially when two or more threads need to finish before you can continue.

Killing Threads

To terminate a thread, the simplest option available is to use the `pthread_cancel` function.

```
function pthread_cancel(ThreadID: TThreadID): Integer; cdecl;
```

The `pthread_cancel` function sends a cancellation request to the thread. By default, when the `pthread_cancel` function is sent to a thread, the thread is killed. However, this is not always the case. Similar to the way a process can block a signal, a thread can choose to ignore or defer termination if it chooses to use the `pthread_setcancelstate` and the `pthread_setcanceltype` functions.

```
const
    PTHREAD_CANCEL_ENABLE = 0;
    PTHREAD_CANCEL_DISABLE = 1;

    PTHREAD_CANCEL_DEFERRED = 0;
    PTHREAD_CANCEL_ASYNCHRONOUS = 1;
```

```
function pthread_setcancelstate(State: Integer; OldState: PInteger): Integer; cdecl;
function pthread_setcanceltype(CancelType: Integer; OldType: PInteger): Integer; cdecl;
procedure pthread_testcancel; cdecl;
```

The `pthread_setcancelstate` function sets whether or not the current thread can be canceled. The function takes a single integer that defines whether or not the thread can be canceled. The value that is passed into this function is usually either the `PTHREAD_CANCEL_ENABLE` constant to enable the thread to be canceled or `PTHREAD_CANCEL_DISABLE` constant to not allow the thread to be canceled. The `OldState` parameter in the `pthread_setcancelstate` function is either a pointer to an integer that can hold the thread's previous cancellation state, or a nil pointer to signify that the `OldState` is not required.

The `pthread_setcanceltype` function defines how the current thread will react to a cancellation request. The function takes two parameters that will define how the thread will react to any cancellation request. If the value for the `CancelType` parameter is `PTHREAD_CANCEL_ASYNCHRONOUS`, the moment that a cancellation request is received by the thread, it will terminate. If the value for the `CancelType` parameter is `PTHREAD_CANCEL_DEFERRED`, the function will continue until it reaches something known as a cancellation point.

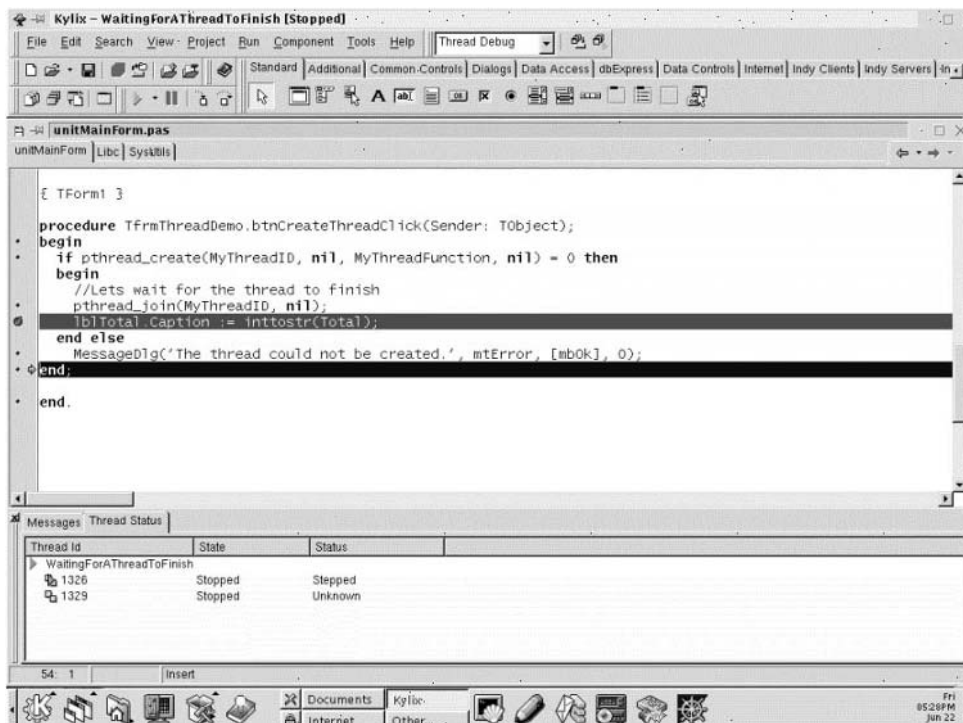
A cancellation point is placed within the thread's execution that calls any of the functions `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `sem_wait`, `sigwait`, or `pthread_testcancel`. These functions are called cancellation points because they internally check to see if the thread has had a cancellation request sent to it, and if so, terminate the thread.

The `pthread_testcancel` function is especially useful here. Its only purpose is to act as a cancellation point. If your thread is running without much chance of reaching a cancellation point, the `pthread_testcancel` function can be used to make sure that the thread can respond to a cancellation point.

One thing that should be noted about threads that terminate is that when a thread actually terminates, in the Kylix debugger you may notice that the thread appears to still be active. In fact, when the example from Listing 6.5 is run in the debugger and the thread executes after we have waited for the thread to terminate, it appears that the thread is still

running. The thread that remains is actually a thread created to manage other threads in the process. This is why you will always see a separate thread such as the one shown in Figure 6.4.

Figure 6.4 - Looking at a thread after it has been canceled



It is true at this point that the thread has been canceled, but the thread ID is still active in case other threads attempt to call `pthread_join` on that thread. Regardless, the resources used to create the thread were released when the thread terminated.

Synchronization Methods Using POSIX Threads

Let's consider our prior example of using a thread to tally up some numbers. Let's extend that example to tally up a real resource such as rooms in a hotel. In this example, the function that is called when a thread is created is the one from Listing 6.6.

Listing 6.6 - Our room booking application

```
unit unitMutexExampleForm;

{
    This is a demo for a hotel that books individual rooms.
    This demonstrates the wrong way to access variables that
    will be shared among different threads.

    Running this application is designed to give incorrect results
}
```

```

interface
uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs, Libc,
  QStdCtrls, QComCtrls;

type
  TfrmMutexDemo = class(TForm)
    btnSReserveRooms: TButton;
    mmInfo: TMemo;
    Label1: TLabel;
    seNumberOfThreads: TSpinEdit;
    Label2: TLabel;
    procedure btnSReserveRoomsClick(Sender: TObject);
  public

  end;

var
  frmMutexDemo: TfrmMutexDemo;

implementation

{$R *.xfm}

var
  ThreadIDs: array[1..5] of Cardinal;
  RoomsTaken, RoomsLeft: integer;
  Rooms: array[1..1000] of boolean;

function BookSomeRooms(Data: Pointer): integer; cdecl;
var
  counter: integer;
  RoomNo: integer;
begin
  for counter := 1 to 1000 do
  begin
    RoomNo := random(1000) + 1;
    if not Rooms[RoomNo] then
    begin
      //Take the room
      Sleep(5); //This is to emulate some task that
               //may take a while
      inc(RoomsTaken);
      Rooms[RoomNo] := true;
    end;
  end;

  Result := 0;
end;

procedure TfrmMutexDemo.btnSReserveRoomsClick(Sender: TObject);
var
  counter: integer;
begin
  mmInfo.Lines.Clear;
  mmInfo.Lines.Add('Setting the Rooms taken to 0');

```

```
//Set the default values
for counter := 1 to 1000 do
    Rooms[counter] := false;
RoomsTaken := 0;
RoomsLeft := 0;

//Create the threads
mmInfo.Lines.Add('Creating the threads');
for counter := 1 to seNumberOfThreads.Value do
    pthread_create(ThreadIDs[counter], nil, BookSomeRooms, nil);

//Wait for the threads to finish
mmInfo.Lines.Add('Waiting for the threads to finish');
for counter := 1 to seNumberOfThreads.Value do
    pthread_join(ThreadIDs[counter], nil);

//Write the number of rooms that were taken
mmInfo.Lines.Add('There were '+inttostr(RoomsTaken)+' rooms taken.');
```

//Let's see how many rooms are left.

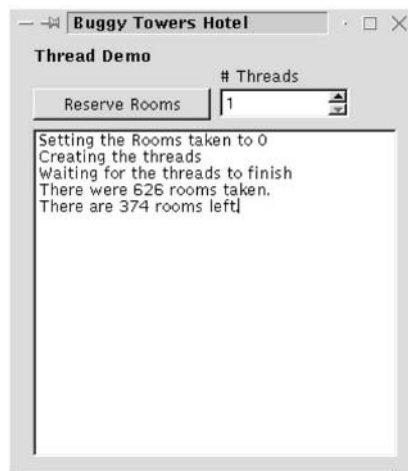
```
for counter := 1 to 1000 do
    if not Rooms[counter] then
        inc(RoomsLeft);

mmInfo.Lines.Add('There are '+inttostr(RoomsLeft)+' rooms left.');
```

end;

```
initialization
    Randomize;
end.
```

Figure 6.5 -
Using a single
thread in the
room booking
application



BookSomeRooms in Listing 6.6 is effective in booking rooms when one thread is running, as shown in Figure 6.5. The number of rooms allocated and the number of rooms will always return a value of 1,000 when a single thread is used.

Figure 6.6 -
Using multiple
threads in the
room booking
application



When multiple threads are used, however, as demonstrated in Figure 6.6, the total number of rooms booked and the number of rooms taken total 1,004. This total comes as a result of timing issues between multiple threads. The problem code that is causing this issue is contained within the `BookSomeRooms` function. The specific code that causes the problem is shown in Listing 6.7.

Listing 6.7 - The timing problem with the threaded function

```
if not Rooms[RoomNo] then
begin
    //Take the room
    Sleep(5); //This is to emulate some task that
              //may take a while
    inc(RoomsTaken);
    Rooms[RoomNo] := true;
end;
```

The problem with the function is that between the time the room is checked and the room is actually taken, another thread may take the room also. It is for this reason that there is an inconsistent number of rooms booked. This inconsistency can be resolved by making sure the action of booking the room is an atomic unit so that no timing issues can disrupt the process of booking a room.

The mechanism provided by the POSIX standard is to use a mutex. Mutex is short for mutual exclusion and gives the ability for a thread to have exclusive access to data or some resource, so that the data or resource cannot be accessed by more than one thread at a time.

In the hotel example in Listing 6.6, a mutual exclusion can be created to block other threads from changing the values of the `Rooms` array, so that rooms would not be booked twice. Listing 6.8 demonstrates how the hotel example can be modified to avoid the problem of double booking.

Listing 6.8 - Protecting the resource with a mutex

```
unit unitMutexExampleForm;

{
```

```

        This is a demo for a hotel that books individual rooms.
        This demonstrates the correct way to access variables that
        will be shared among different threads using mutexes.
    }

interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs, Libc,
    QStdCtrls, QComCtrls;

type
    TfrmMutexDemo = class(TForm)
        btnSReserveRooms: TButton;
        mmInfo: TMemo;
        Label1: TLabel;
        seNumberOfThreads: TSpinEdit;
        Label2: TLabel;
        procedure btnSReserveRoomsClick(Sender: TObject);
    public

        end;

var
    frmMutexDemo: TfrmMutexDemo;

implementation

{$R *.xfm}

var
    ThreadIDs: array[1..5] of Cardinal;
    RoomsTaken, RoomsLeft: integer;
    Rooms: array[1..1000] of boolean;
    BookingRoomMutex: pthread_mutex_t;
    EmptyMutexAttribute: pthread_mutexattr_t;

function BookSomeRooms(Data: Pointer): integer; cdecl;
var
    counter: integer;
    RoomNo: integer;
begin
    for counter := 1 to 1000 do
        begin
            RoomNo := random(1000) + 1;

            pthread_mutex_lock(BookingRoomMutex);
            try
                if not Rooms[RoomNo] then
                    begin
                        //Take the room
                        Sleep(5); //This is to emulate some task that
                                //may take a while
                        inc(RoomsTaken);
                        Rooms[RoomNo] := true;
                    end;
            finally

```

```

        pthread_mutex_unlock(BookingRoomMutex);
    end;
end;

Result := 0;
end;

procedure TfrmMutexDemo.btnSReserveRoomsClick(Sender: TObject);
var
    counter: integer;
begin
    mmInfo.Lines.Clear;
    mmInfo.Lines.Add('Setting the Rooms taken to 0');

    //Set the default values
    for counter := 1 to 1000 do
        Rooms[counter] := false;
        RoomsTaken := 0;
        RoomsLeft := 0;

        //Create the threads
        mmInfo.Lines.Add('Creating the threads');
        for counter := 1 to seNumberOfThreads.Value do
            pthread_create(ThreadIDs[counter], nil, BookSomeRooms, nil);

            //Wait for the threads to finish
            mmInfo.Lines.Add('Waiting for the threads to finish');
            for counter := 1 to seNumberOfThreads.Value do
                pthread_join(ThreadIDs[counter], nil);

            //Write the number of rooms that were taken
            mmInfo.Lines.Add('There were '+inttostr(RoomsTaken)+' rooms taken.');
```

//Let's see how many rooms are left.

```

        for counter := 1 to 1000 do
            if not Rooms[counter] then
                inc(RoomsLeft);

        mmInfo.Lines.Add('There are '+inttostr(RoomsLeft)+' rooms left.');
```

end;

initialization

```

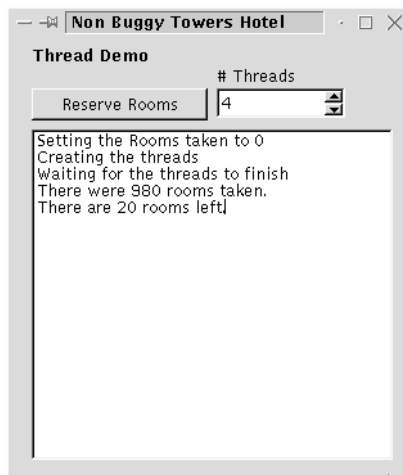
    Randomize;
    //Create the Mutex Attribute
    pthread_mutexattr_init(EmptyMutexAttribute);
    //Create the Mutex
    pthread_mutex_init(BookingRoomMutex, EmptyMutexAttribute);
finalization
    //Free the Mutex Attribute
    pthread_mutexattr_destroy(EmptyMutexAttribute);
    //Free the Mutex
    pthread_mutex_destroy(BookingRoomMutex);
end.
```

The mutex works by attempting to lock the resource when the thread is executing the line:

```
pthread_mutex_lock(BookingRoomMutex);
```

The first thread that calls this function will be the only thread at this stage that can move to the following statement. Other threads that reach this point will not be able to proceed beyond this point until the mutex has been released with a call to `pthread_mutex_unlock` and that thread has been given the mutex. As a result, the code between the call to `pthread_mutex_lock` and `pthread_mutex_unlock` will only ever be executed by one thread at a time, effectively controlling the resource. Figure 6.7 demonstrates the resulting output from the application once the example is modified to use a mutex. As you can see, the number of rooms remain consistent.

Figure 6.7 -
The hotel
application
when using a
mutex



The mutex is defined by the `pthread_mutex_t` data type. Before any thread can begin trying to lock the mutex, the mutex should first be initialized with a call to the `pthread_mutex_init` function. In this demo application, the mutex is initialized in the initialization section of the unit, and deallocated within the finalization section using the `pthread_mutex_destroy`.

The `pthread_mutex_init` function is declared in `LibC.pas` as

```
function pthread_mutex_init(var Mutex: TRTLCriticalSection;  
    var Attr: TMutexAttribute): Integer; cdecl;
```

The first parameter is the `pthread_mutex_t` variable that will be used to identify the mutex. The second parameter is the variable of `TMutexAttribute`, which defines the attributes for the mutex. This parameter should have been a `PMutexAttribute` data type so that `nil` could have been passed into this parameter to state that the default attributes for this mutex should be used. Otherwise, the mutex attribute should be declared and initialized using the `pthread_mutexattr_init` function as shown in Listing 6.8.

Like many synchronization techniques, the mutex is a convention that needs to be followed uniformly throughout the application for it to work. The variables in the section of code that is protected by the mutex cannot be altered by another thread.

Not only are mutexes helpful when protecting data, but they can also be used to protect resources such as a device that can only communicate with one thread at a time. Because

the mutex only protects the section of code between the lock and unlock, a mutex can be used on any resource that requires only one thread have access to the data.

Mutexes are effective methods of providing protection to a single resource, but when multiple resources need to be managed, you may need to look at managing them with semaphores.

Semaphores

In Chapter 5, when looking at interprocess communication, we introduced semaphores as a way of managing resources between processes. Now that you are working with threads rather than processes, the rules stay the same. To get a thorough understanding of how semaphores operate, see the section on semaphores in Chapter 5.

The semaphores presented in Chapter 5 are known as System V semaphores, and are mainly used on UNIX-based operating systems for managing resources between processes. The semaphores used in POSIX-threaded applications come from the POSIX standard and are the best method of managing resources between threads.

Semaphores are a great way to manage the allocation of resources, except that when working with threads, POSIX 1003.4 has its own standard set of functions for using semaphores rather than using the `semget`, `semop`, and `semctl` functions. The POSIX 1003.4 semaphore functions are shown in Listing 6.9.

Listing 6.9 - Using the POSIX 1003.4 semaphore functions

```
function sem_init(var __sem: TSemaphore; __pshared: LongBool;
  __value: LongWord): Integer; cdecl;
function sem_destroy(var __sem: TSemaphore): Integer; cdecl;
function sem_open(__name: PChar; __oflag: Integer): PSemaphore; cdecl; varargs;
function sem_close(var __sem: TSemaphore): Integer; cdecl;
function sem_unlink(__name: PChar): Integer; cdecl;
function sem_wait(var __sem: TSemaphore): Integer; cdecl;
function sem_timedwait(var __sem: TSemaphore; const __abstime: timespec): Integer;
  cdecl;
function sem_trywait(var __sem: TSemaphore): Integer; cdecl;
function sem_post(var __sem: TSemaphore): Integer; cdecl;
function sem_getvalue(var __sem: TSemaphore; var __sval: Integer): Integer; cdecl;
```

Within the POSIX standard there are two different types of semaphores: named semaphores and unnamed semaphores. Unnamed semaphores exist only in memory on the system. Unnamed semaphores will live in shared memory if they are to be used for management of resources, but in the case where they are being used for a threaded application, the semaphore will live in the memory space of the process only. Unnamed semaphores are created with the `sem_init` function and destroyed with the `sem_destroy` function.

On the other hand, the POSIX specification states that named semaphores are created with the `sem_open` function and closed with the `sem_close` function, and the first parameter defines the name of the semaphore that will be used. This name begins with a forward slash “/” and then represents the location of the semaphore that is used, so that it can be used between processes. Unfortunately, named semaphores are not available in LinuxThreads and calling the named semaphore functions `sem_open` and `sem_close` will result in the `ENOSYS` error code being returned. If you want to work with semaphores across process boundaries, it is a better idea that you use the System V semaphores shown in Chapter 5.

Working with semaphores is easy enough. First you will initialize a semaphore with the number of resources that the semaphore is managing. Then threads will try to gain access to the semaphore resource. This is done using the `sem_init` function. The `sem_init` function takes a `sem_t` variable, which is the semaphore variable itself, whether or not the semaphore is to be shared between processes, which is always false on Linux, and finally, the number of resources to protect.

```
var
    ThreadPoolSemaphore: sem_t;

...
begin
    //Create the semaphore
    sem_init(ThreadPoolSemaphore, false, 5);
```

Obtaining a lock on a semaphore is done by using the `sem_wait` function. This function will block until a semaphore resource is available for the current thread to obtain. Once the semaphore is obtained, the function continues on normally.

```
    //Wait for a semaphore to become available
    sem_wait(ThreadPoolSemaphore);
```

After the processing on the semaphore is done, the resource can be released back to the semaphore with the `sem_post` function.

```
    //We should release the semaphore now as we have finished with it.
    sem_post(ThreadPoolSemaphore);
```

One of the main uses of semaphores is to control a pool of resources. Protecting pools of resources is mainly done within server applications, such as a socket service like a Web server. Listing 6.10 contains a simple Web server that manages connections by spawning a new process every time a client connects and requests a page. This example will be modified so that the server calls on an existing pool of threads to connect to the page.

This will be accomplished by creating the threads before anything else is done. Every thread will operate on a single request and when that request has been handled, will wait for the next request and then handle that request and so on. There will be a fixed number of threads created to manage the connections. In this case, the threads that we are using to handle the requests are the resources that we want to protect. When a request is retrieved, the application determines if there is a thread available that is not handling any requests. The semaphore is used to protect the resource by initializing the semaphore value to the number of threads in the thread pool. When a request comes in, the application will block until a semaphore resource becomes available. This way only the number of threads that are created are used in the application.

Listing 6.10 - A Web server using thread pooling

```
program SimpleWebServer;

(*
    This program is a simple Web application that
    accepts any Web request and returns the contents
    of a file only if it is .html, .htm or .txt
```

The application in the sockets chapter uses forking to

allow for multiple connections.
 This application uses POSIX threads to handle the HTTP clients
 and uses a semaphore to manage the active/non-active threads
 so that pooling can be implemented.

Conditional Variables are used to awaken the threads as
 POSIX does not really implement a suspend/resume mechanism

```
*)

{$APPTYPE CONSOLE}

uses
  SysUtils, Classes, Types, Libc;

var
  //Server Variables
  ServerSocketID: integer;
  ServerAddressToBindTo: TSocketAddrIn;

  //Client Socket Variables
  NewClientConnection: integer;
  SizeOfClientAddress: integer;

  //Misc
  counter: integer;

  //Functions used for threading
  Threads: array[0..4] of pthread_t;
  ThreadFileDescriptors: array[0..4] of integer;
  ThreadAllocationMutex: pthread_mutex_t;
  ThreadAllocationMutexAttr: pthread_mutexattr_t;

  ThreadPoolSemaphore: sem_t;
  ThreadConditionVars: array[0..4] of pthread_cond_t;
  ThreadConditionVarMutexes: array[0..4] of pthread_mutex_t;

function CalculateWebResponse(strWebRequest: string): string;
var
  Info: TStringList;
  DataToReturn: TStringList;
  counter: integer;
  strFilename: string;
  strErrorMessage: string;
  iPos: integer;
  strFileExt: string;
  WebFileDirectory: string;
begin
  Result := '';

  DataToReturn := TStringList.Create;
  try
    Info := TStringList.Create;
    try
      Info.Text := strWebRequest;
      //There should be a line that is of the format GET /filename.htm HTTP/1.0
      //That is what we look for, otherwise we will return an error to the client
      strFilename := '';
```

```

for counter := 0 to Info.Count - 1 do
begin
  iPos := Pos('GET ', Info[counter]);
  if iPos <> 0 then
  begin
    //Here we get the file that we want to return.
    strFilename := Copy(Info[counter], iPos + 4, Length(Info[counter]));
    //The string will now look like "/filename.htm HTTP/1.0"
    //so we only have to remove the HTTP to get the filename
    iPos := Pos('HTTP', strFilename);
    if iPos <> 0 then
      strFilename := Trim(Copy(strFilename, 1, iPos-1))
    else begin
      strFilename := '';
      strErrorMessage := 'The file could not be found.';
    end;

    break;
  end;
end;

//Calculate the Web directory
WebFileDirectory := ExtractFilePath(ParamStr(0));
if Copy(WebFileDirectory, Length(WebFileDirectory), 1) <> '/' then
  WebFileDirectory := WebFileDirectory + '/';
WebFileDirectory := WebFileDirectory + 'webfiles/';

//Add support for the index.html as default
if strFilename = '/' then
  strFilename := '/index.html';

//Remove the first forward slash as this is appended to directory anyway
//Otherwise the system will be unable to find the file
strFilename := Copy(strFilename, 2, Length(strFilename));

//Let's see if the file exists first
if not FileExists(WebFileDirectory+strFilename) then
begin
  //the file does not exist
  strFilename := '';
  strErrorMessage := 'The file could not be found.';
end else begin
  strFileExt := lowercase(ExtractFileExt(strFilename));
  if not ((strFileExt = '.htm') or (strFileExt = '.html') or
    (strFileExt = '.txt')) then
  begin
    //The file extension is invalid. Return an error
    strErrorMessage := 'Invalid File Extension Type.';
    strFilename := '';
  end;
end;

if strFilename <> '' then
begin
  //We can return the file that was requested back to the user
  writeln('Returning the web file: ', strFilename);
  DataToReturn.LoadFromFile(WebFileDirectory+strFilename);

```



```

        DataToReturn.Insert(0, 'HTTP/1.0 200 OK');
        DataToReturn.Insert(1, '');
    end else begin
        //We will return the File not File error message
        writeln('A Request was made for a file that does not exist.');
```

DataToReturn.Add('HTTP/1.0 404 Not Found');
 DataToReturn.Add('');
 DataToReturn.Add('<html><body><h1>HTTP/1.0 404 Not Found</h1><p>');
 DataToReturn.Add('The file you requested is not located on this Web server ');
 DataToReturn.Add('or is not a html or text file. This simple Web server for ');
 DataToReturn.Add('the "Tomes of Kylix - Linux API" only deals with HTML and ');
 DataToReturn.Add('text files only.<p>');
 DataToReturn.Add('Error: '+strErrorMessage+'</body></html>');
 end;
finally
 Info.Free;
end;

DataToReturn.Add('');
DataToReturn.Add('');

Result := DataToReturn.Text;
finally
 DataToReturn.Free;
end;
end;

```

function ProcessWebRequestThread(Data: Pointer): integer; cdecl;
var
    ThreadPosition: integer;
    MyBuffer: array[0..10000] of char;
    ResponseToSend: string;
    ResponseData: PChar;
    semValue: integer;
begin
    //This function is run within the context of an individual thread
    //and handles an individual request and waits until the data is ok.
    //The function uses conditional variables to determine when to wake up.
    ThreadPosition := integer(Data);
    repeat
        //We need to wait for the Conditional Variable to be activates
        //to notify us of when a thread is active.
        pthread_cond_wait(ThreadConditionVars[ThreadPosition],
            ThreadConditionVarMutexes[ThreadPosition]);

        //Then we handle the process and do it all again
        recv(ThreadFileDescriptors[ThreadPosition], MyBuffer, 10000, 0);
        ResponseToSend := CalculateWebResponse(MyBuffer);

        //Send the Response back to the Web server
        ResponseData := StrAlloc(Length(ResponseToSend)+1);
        try
            StrPCopy(ResponseData, ResponseToSend);
            send(NewClientConnection, ResponseData^, StrLen(ResponseData)+1, 0);
        finally
            StrDispose(ResponseData);
        end;
    end;
end;

```

```

    __close(NewClientConnection);
end;

ThreadFileDescriptors[ThreadPosition] := 0;

//Let's add a termination point to the thread just in case.
pthread_testcancel;

//We should release the semaphore now as we have finished with it.
sem_post(ThreadPoolSemaphore);
sem_getvalue(ThreadPoolSemaphore, semValue);
writeln(semValue, ' threads are being used in the thread pool.');
```

until False;

```

    Result := 0;
end;

begin
    //Let's create the resources required for the thread pool
    //Create the Mutex Attribute
    pthread_mutexattr_init(ThreadAllocationMutexAttr);
    pthread_mutexattr_setpshared(ThreadAllocationMutexAttr, PTHREAD_PROCESS_PRIVATE);
    //Create the Mutex
    pthread_mutex_init(ThreadAllocationMutex, ThreadAllocationMutexAttr);

    //Create the semaphore
    sem_init(ThreadPoolSemaphore, false, 5);

    //Create the conditional Variables
    //The conditional variables are what will be used to wake up the
    //threads and tell them that they have a request waiting
    for counter := 0 to 4 do
    begin
        pthread_cond_init(ThreadConditionVars[counter], nil);
        pthread_mutex_init(ThreadConditionVarMutexes[counter], ThreadAllocationMutexAttr);
    end;

    //Now finally, let's create the threads
    for counter := 0 to 4 do
    begin
        pthread_mutex_lock(ThreadConditionVarMutexes[counter]);
        pthread_create(Threads[counter], nil, ProcessWebRequestThread, Pointer(counter));
    end;

    //Create the Socket
    ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);
    try
        //Define the Address to Connect to
        with ServerAddressToBindTo do
        begin
            sa_family := AF_INET;
            sin_addr.S_addr := htonl(INADDR_ANY);
            sin_port := htons(8081);
        end;

        //Bind the Socket

```

```

    if bind(ServerSocketID, ServerAddressToBindTo, sizeof(ServerAddressToBindTo)) = -1
then
    begin
        perror('Could not bind the address.');
```

Exit;

```
    end;
```

//Accept Connections to the Socket

```
    if listen(ServerSocketID, 10) = -1 then
begin
    perror('Could not listen on the socket.');
```

Exit;

```
end;
```

//Get any new connection, fork, process the client in the forked child
//and then close the forked process

```
writeln('Server Listing.....');
```

repeat

```
    SizeOfClientAddress := sizeof(TSockAddrIn);
    NewClientConnection := accept(ServerSocketID, @NewClientConnection,
                                @SizeOfClientAddress);
```

//When we get here, then we have a new socket connection, so we fork and handle
//the processing in the forked process

```
writeln('Received a new Web Request. Obtain a thread from the thread pool and
        allocate it');
```

//Wait for a semaphore to become available

```
sem_wait(ThreadPoolSemaphore);
```

//Now that we have access to a semaphore, let's assign it to a thread

```
pthread_mutex_lock(ThreadAllocationMutex);
```

try

```
    //Allocate the appropriate thread
    for counter := 0 to 4 do
begin
    if ThreadFileDescriptors[counter] = 0 then
begin
        ThreadFileDescriptors[counter] := NewClientConnection;
```

//Now we need to wake up the thread using the conditional variable

```
        pthread_cond_signal(ThreadConditionVars[counter]);
        break;
```

```
    end;
```

```
end;
```

```
finally
    pthread_mutex_unlock(ThreadAllocationMutex);
end;
```

until false;

```
finally
    //Close the server socket.
    __close(ServerSocketID);
end;
```

end.

The threads actually wait using a mechanism called a condition variable, which is covered in the next section. For developers coming from a Windows background, POSIX threads do not have the ability to suspend and restore a thread in the same way Windows does. As a result, we use condition variables, which wait for some signal to be sent so that the thread can wake up.

Condition Variables

Sometimes in a multi-threaded application you may want to suspend the current thread until some event occurs that you want the thread to deal with. Notice that this is exactly what was done in Listing 6.10. A thread that handles a Web request is created in advance and then waits for a request for a Web page. Normally, you will use condition variables when the current thread cannot proceed until some event has occurred. For example, if a thread is collating data from several sources, a condition variable may be signaled when new data is available.

This is the true power of condition variables and is what they were intended to do. Condition variables are not designed to protect access to a resource like a mutex or a semaphore, but instead to notify other threads of some occurrence a thread may be interested in.

Listing 6.11 - Using the condition variable data types

```
Type
pthread_cond_t = packed record
    __c_lock: _pthread_fastlock;
    __c_waiting: _pthread_descr;
end;
TCondVar = pthread_cond_t;
PCondVar = ^TCondVar;
```

To use condition variables in your application, you must declare a `pthread_cond_t` data type variable which represents the condition variable. Once the variable is declared, you then initialize the condition variable for use within the application. Like most variables used for synchronization in POSIX threads, condition variables have their own initialization function, `pthread_cond_init`.

Listing 6.12 - Using the condition variable functions

```
function pthread_cond_init(var Cond: TCondVar;
    var CondAttr: TPthreadCondattr): Integer; cdecl;
function pthread_cond_destroy(var Cond: TCondVar): Integer; cdecl;
function pthread_cond_signal(var Cond: TCondVar): Integer; cdecl;
function pthread_cond_broadcast(var Cond: TCondVar): Integer; cdecl;
function pthread_cond_wait(var Cond: TCondVar;
    var Mutex: TRTLCriticalSection): Integer; cdecl;
function pthread_cond_timedwait(var Cond: TCondVar;
    var Mutex: TRTLCriticalSection; const AbsTime: TTimeSpec): Integer; cdecl;
```

Listing 6.13 - Initializing a condition variable

```
var
    ThreadConditionVar: pthread_cond_t;
```

```

Begin
    //initialize the condition variable
    pthread_cond_init(&ThreadConditionVar, nil);

    //...
end;

```

Once the condition variable is initialized, you specify that you want to wait until the condition variable is signaled. Waiting for a condition variable to be signaled is done via the `pthread_cond_wait` function. This function takes two parameters, the first being the condition variable that will be signaled and also a POSIX mutex that is currently locked by the current thread. Using the `pthread_cond_wait` function is shown in Listing 6.14. The `pthread_cond_timedwait` can also be used to wait for a condition variable to be signaled, but this function will return if the timeout period is not reached. In the waiting functions, the mutex is used as a parameter when different condition variables are used in an application so that, at most, only one condition variable will be processed at a time.

Listing 6.14 - Waiting for a condition variable to be signaled

```

//Wait for the Condition Variable to be signaled
pthread_cond_wait(&ThreadConditionVar, MyMutex);
//The Condition Variable has now been signaled

```

So now that you know how to wait for a condition variable, the last thing that you will need to work with is to actually signal the condition variable. This is done by using the `pthread_cond_signal` function or the `pthread_cond_broadcast` function. Both of these functions will signal a condition variable, but the `pthread_cond_signal` function will only signal one thread that is waiting on the condition variable. When you call this function only one thread that is waiting on the condition will be woken, and you can never be sure which thread it is going to be.

Listing 6.15 - Signaling a condition variable

```

//Signal the condition variable
pthread_cond_signal(&ThreadConditionVar);

```

If you need all the threads that are waiting on a single condition variable to be signaled, you should use the `pthread_cond_broadcast` function. Using this function will result in waking up all threads that are waiting on a single condition variable to be signaled. An example of this is shown in Listing 6.16.

Listing 6.16 - Broadcasting to a condition variable

```

//Signal all threads waiting on the condition variable
pthread_cond_broadcast(&ThreadConditionVar);

```

The signaling mechanism of condition variables provide a good fit into any applications that need notification of some event that occurs. In Listing 6.10, a multi-threaded Web server used condition variables to notify a thread that a new Web page request had been sent through. In this way, the thread can wake up, process the request, and then wait for that condition variable to be notified again.



Note: It is worth noting that in this section on condition variables, the term “signal” refers to notification and not to the concept of signals discussed in Chapter 5.

Listen Carefully, I Shall Say This Only `pthread_once`

Much of the data we use relies on synchronization data types such as the mutex and semaphore, but these data types need to be initialized before they can be used by the thread functions. There are two effective ways to manage the resources when a threaded application is being run.

The first is to make sure all the synchronization variables are initialized before any of the threads are created. This will work effectively in most cases where the number of resources to protect are known at compile time. In the case where the resources that you are protecting are not known until run time, you may need to use the `pthread_once` function shown in Listing 6.17.

Listing 6.17 - The `pthread_once` function

```
Const
    PTHREAD_ONCE_INIT = 0;

type
    TInitOnceProc = procedure; cdecl;

function pthread_once(var InitOnceSemaphore: Integer;
    InitRoutine: TInitOnceProc): Integer; cdecl;
```

The `pthread_once` function takes two parameters. The first is a `pthread_once_t` data type, which is used to determine if the second parameter, `InitRoutine`, has been called or not. The second parameter is a procedure with no parameters that uses the `cdecl` calling convention. This is the routine that is guaranteed to be executed only once.

Like most synchronization techniques used by POSIX, the `pthread_once` function relies on blocking in order to work. The first thread that calls the function will cause the function to execute only if it has not been executed already. Other threads that then make a call to the `pthread_once` function will block if the function passed to `pthread_once` is still being executed, or will continue if the function has already executed.

By doing this when initializing variables, each thread can be guaranteed the resources that the threading function will access will be there. If we modify our previous example for hotels, we can use the `pthread_once` function to execute the code to initialize the mutex.

Listing 6.18 - Using the `pthread_once` function

```
var
    ThreadIDs: array[1..5] of Cardinal;
    RoomsTaken, RoomsLeft: integer;
    Rooms: array[1..1000] of boolean;
    BookingRoomMutex: pthread_mutex_t;
    EmptyMutexAttribute: pthread_mutexattr_t;
    RoomsInitOnce: pthread_once_t = PTHREAD_ONCE_INIT;
```

```

procedure BookingRoomsInit; cdecl;
begin
    //Here we initialize the data
    //Create the Mutex Attribute
    pthread_mutexattr_init(EmptyMutexAttribute);
    //Create the Mutex
    pthread_mutex_init(BookingRoomMutex, EmptyMutexAttribute);
end;

function BookSomeRooms(Data: Pointer): integer; cdecl;
var
    counter: integer;
    RoomNo: integer;
begin
    //Let's initialize the data that is required
    pthread_once(RoomsInitOnce, BookingRoomsInit);

    for counter := 1 to 1000 do
    begin
        RoomNo := random(1000) + 1;

        pthread_mutex_lock(BookingRoomMutex);
        try
            if not Rooms[RoomNo] then
            begin
                //Take the room
                Sleep(5); //This is to emulate some task that
                        //may take a while
                inc(RoomsTaken);
                Rooms[RoomNo] := true;
            end;
        finally
            pthread_mutex_unlock(BookingRoomMutex);
        end;
    end;

    Result := 0;
end;

```

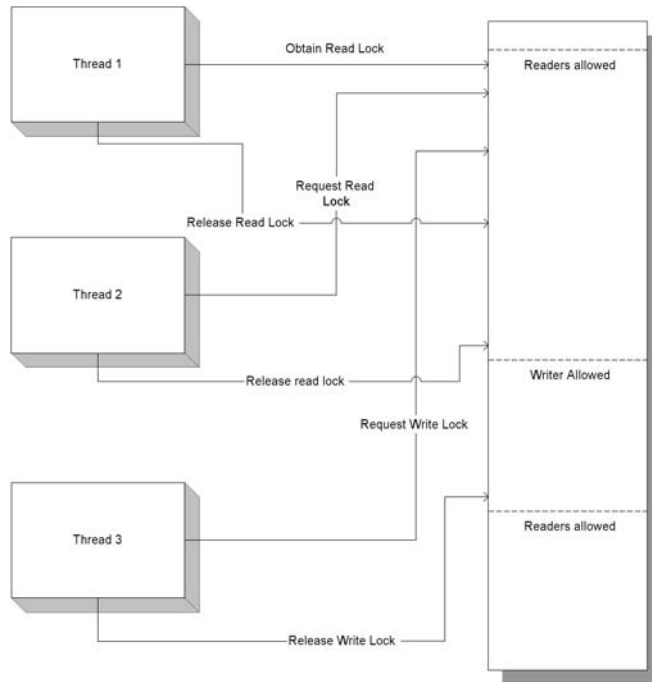
Other Synchronization Methods

The POSIX standards that are used for threading and synchronization have more techniques for synchronization that can be used within your Linux applications. In the majority of cases, mutexes, semaphores, and condition variables will be most of the synchronization that you need. There are other methods that you are unlikely to use, such as read/write locks, spin locks, and barriers.

Read/write locks are useful for data that is read more often than it is modified. It is used where data can be read by any thread, but can only be written to by one thread. This kind of lock would be suitable for a database style application where in many cases the data is to be read by many threads, but only updated by one thread at a time. Like most synchronization methods, read/write locks allow a single thread to access the resource. If there are read locks on the data, then the write lock will occur after the read locks have been released. Any locks that are requested once the write lock has been requested will be blocked until the write lock has finished its work as shown in Figure 6.8. In this way, read/write locks

behave like mutexes when a write lock is requested. Using read/write locks is more efficient than using mutexes as a mutex will block all other threads, where the read/write lock will block only when the thread is writing. As fewer threads are blocked, more work can then be done concurrently.

Figure 6.8 -
The read/
write lock



Listing 6.19 demonstrates the use of the read/write lock by creating a simple application that has a thread that can read information about an account, another thread that can modify account balances if the thread is overdrawn, and another thread that calculates interest on the account. This application demonstrates the different methods of obtaining both read and write locks from different threads.

Listing 6.19 - Using a read/write lock

```
unit unitBankAccountDisplayForm;

{
    This application demonstrates the use of read/write locks
    by having separate threads perform different operations on records.
    The main thread (UI) can access the records, one thread looks for
    negative balance, while another thread calculates interest.

    This demonstration also demonstrates that the various
    synchronization methods can be used with the TThread class
```


This demonstration does not access records, it is used to demonstrate the principles behind reader/writer locks.

Written for the "Tomes of Kylix - The Linux API" by Glenn Stephens

```

}

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls, QComCtrls, Libc;

type
  TfrmAccounts = class(TForm)
    Label1: TLabel;
    lvAccounts: TListView;
    btnGetAccountData: TButton;
    edAccountNo: TEdit;
    Label2: TLabel;
    btnCheckAccountAccounts: TButton;
    btnCalculateInterest: TButton;
    Label3: TLabel;
    procedure btnGetAccountDataClick(Sender: TObject);
    procedure btnCheckAccountAccountsClick(Sender: TObject);
    procedure btnCalculateInterestClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure SetUpAccounts;
  public
  end;

  //This structure will be used to hold the account information
  TAccountData = record
    ReadWriteLock: pthread_rwlock_t;
    AccountNumber: integer;
    AccountName: string[20];
    AccountBalance: Currency;
  end;

  TNegativeAccountCheckThread = class(TThread)
  protected
    procedure Execute; override;
  end;

  TInterestOnAccountsThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  frmAccounts: TfrmAccounts;
  Accounts: array[1..1000] of TAccountData;

implementation

{$R *.xfm}

```

```

{ TfrmAccounts }

procedure TfrmAccounts.SetUpAccounts;
var
  counter: integer;
begin
  //This function sets up the accounts with random dummy data
  for counter := 1 to 1000 do
  begin
    //Set up the account
    pthread_rwlock_init(Accounts[counter].ReadWriteLock, nil);
    Accounts[counter].AccountNumber := counter;
    Accounts[counter].AccountName := 'Account #' + inttostr(counter);
    Accounts[counter].AccountBalance := random(1000) - 200;
  end;
end;

procedure TfrmAccounts.btnGetAccountDataClick(Sender: TObject);
var
  AccountItem: TListItem;
  AccountNo: integer;
begin
  AccountNo := strtoint(edAccountNo.Text);
  //Try to find the item in the list view.
  AccountItem := lvAccounts.FindCaption(0, inttostr(AccountNo), false, false, true);
  if AccountItem = nil then
  begin
    AccountItem.Selected := true;
    AccountItem.MakeVisible(false);
  end else begin
    //Let's make an entry in the list view
    AccountItem := lvAccounts.Items.Add;
    AccountItem.Caption := inttostr(AccountNo);
    AccountItem.SubItems.Add('');
    AccountItem.SubItems.Add('');
  end;

  //Try to obtain a read lock on the data so that it can be displayed.
  if pthread_rwlock_tryrdlock(Accounts[AccountNo].ReadWriteLock) = EBUSY then
  begin
    //Now we can display the data about the account
    AccountItem.SubItems[0] := Accounts[AccountNo].AccountName;
    AccountItem.SubItems[1] := CurrToStr(Accounts[AccountNo].AccountBalance);

    //Now let's release the lock
    pthread_rwlock_unlock(Accounts[AccountNo].ReadWriteLock);
  end else
    MessageDlg('The account is currently being updated. Please try again later.',
      mtInformation, [mbOk], 0);
end;

procedure TfrmAccounts.btnCheckAccountAccountsClick(Sender: TObject);
begin
  //This scans the accounts for ones with negative balance and
  //charges them an account keeping fee

```

```

    TNegativeAccountCheckThread.Create(false);
end;

procedure TfrmAccounts.btnCalculateInterestClick(Sender: TObject);
begin
    //This scans the accounts and calculates the interest on accounts
    //with more than $100.
    TInterestOnAccountsThread.Create(false);
end;

{ TNegativeAccountCheckThread }

procedure TNegativeAccountCheckThread.Execute;
var
    counter: integer;
    Balance: Currency;
begin
    //This thread moves through the accounts and looks for accounts with
    //a negative balance and then changes them a $10 fee.

    for counter := 1 to 1000 do
    begin
        //Get a read lock on the Account
        pthread_rwlock_rdlock(Accounts[counter].ReadWriteLock);
        Balance := Accounts[counter].AccountBalance;
        pthread_rwlock_unlock(Accounts[counter].ReadWriteLock);

        if Balance < 0 then
        begin
            //We need to modify the account balance, so we need a write lock
            pthread_rwlock_wrlock(Accounts[counter].ReadWriteLock);
            Accounts[counter].AccountBalance := Accounts[counter].AccountBalance - 10;
            pthread_rwlock_unlock(Accounts[counter].ReadWriteLock);
        end;
    end;
end;

{ TInterestOnAccountsThread }

procedure TInterestOnAccountsThread.Execute;
var
    counter: integer;
    Balance: Currency;
begin
    //This thread moves through the accounts and looks for accounts with
    //a negative balance and then changes them a $10 fee.

    for counter := 1 to 1000 do
    begin
        //Get a read lock on the Account
        pthread_rwlock_rdlock(Accounts[counter].ReadWriteLock);
        Balance := Accounts[counter].AccountBalance;
        pthread_rwlock_unlock(Accounts[counter].ReadWriteLock);

        if Balance > 100.00 then
        begin
            //We need to modify the account balance, so we need a write lock

```

```

pthread_rwlock_wrlock(Accounts[counter].ReadWriteLock);

//Calculate the interest at 10%. (Calculates Daily)
Accounts[counter].AccountBalance := Accounts[counter].AccountBalance +
    (Accounts[counter].AccountBalance * 1/365.25);

pthread_rwlock_unlock(Accounts[counter].ReadWriteLock);
end;
end;
end;

procedure TfrmAccounts.FormCreate(Sender: TObject);
begin
    SetUpAccounts;
end;

end.

```

Spin locks are similar to mutexes in concept, but they are implemented internally to be the quickest way to create a critical section of code, and are mainly used where threads are running on a multiprocessor system. Spin lock API function calls have an interface similar to the mutex, but a spin lock does not have attributes associated with it. It should be noted that under Linux, the spin lock will consistently check the value of the spin lock to obtain the value. Although it is faster than a mutex, it does come at the cost of utilizing more of the CPU.

To see how spin locks are used within an application, see Listing 6.20. Comparing Figure 6.9 and Figure 6.10 will give you insight to the speed difference of both the mutex and the spin lock.

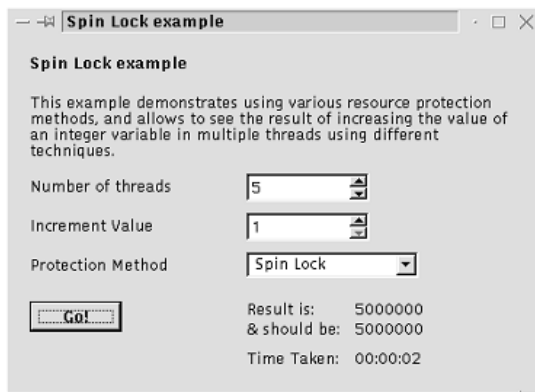


Figure 6.9 - Using spin locks

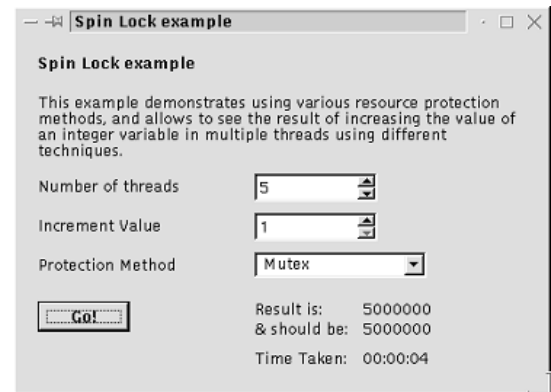


Figure 6.10 - Using mutexes

Listing 6.20 - Using the spin lock

```

unit unitSpinLockForm;

interface

uses

```

SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs, Libc, QStdCtrls, QComCtrls;

```

type
  TfrmSpinLockDemo = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    seThreadCount: TSpinEdit;
    Label4: TLabel;
    seIncrementValue: TSpinEdit;
    Label5: TLabel;
    cbSynchMethod: TComboBox;
    btnGo: TButton;
    lblResult: TLabel;
    lblShouldBeValue: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    lblTimeTaken: TLabel;
    procedure btnGoClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmSpinLockDemo: TfrmSpinLockDemo;

implementation

{$R *.xpm}

var
  ValueToIncrease: double;
  IncrementValue: integer;
  ThreadCount: integer;

  MyMutex: pthread_mutex_t;
  MyMutexAttr: pthread_mutexattr_t;
  MySpinLock: pthread_spinlock_t;

function AddOneToTheCounterValue(Data: Pointer): integer; cdecl;
var
  counter: integer;
begin
  pthread_yield; //Give other threads the chance to be created
  for counter := 1 to 1000000 do
  begin
    //This is the wrong method of doing it. There is no locking of the
    //variable to an individual thread while updating it.
    ValueToIncrease := ValueToIncrease + IncrementValue;
  end;
  Result := 0;
end;
end;

```

```

function AddOneToTheCounterValueMutex(Data: Pointer): integer; cdecl;
var
    counter: integer;
begin
    pthread_yield; //Give other threads the chance to be created
    for counter := 1 to 1000000 do
    begin
        //This is the safer way to do it.
        pthread_mutex_lock(MyMutex);
        ValueToIncrease := ValueToIncrease + IncrementValue;
        pthread_mutex_unlock(MyMutex);
    end;
    Result := 0;
end;

function AddOneToTheCounterValueSpinLock(Data: Pointer): integer; cdecl;
var
    counter: integer;
begin
    pthread_yield; //Give other threads the chance to be created
    for counter := 1 to 1000000 do
    begin
        //This is safe as well, but it uses a spin lock.
        pthread_spin_lock(MySpinLock);
        ValueToIncrease := ValueToIncrease + IncrementValue;
        pthread_spin_unlock(MySpinLock);
    end;
    Result := 0;
end;

procedure TfrmSpinLockDemo.btnGoClick(Sender: TObject);
var
    ThreadFunction: TThreadFunc;
    counter: integer;
    ThreadIDs: array[1..10] of Cardinal;
    StartTime, EndTime: TDateTime;
begin
    //Result the Value
    ValueToIncrease := 0;
    IncrementValue := seIncrementValue.Value;
    ThreadCount := seThreadCount.Value;

    //Allocate and Initialize any variables required.
    case cbSynchMethod.ItemIndex of
        1: begin
            pthread_mutexattr_init(MyMutexAttr);
            pthread_mutex_init(MyMutex, MyMutexAttr);
            ThreadFunction := AddOneToTheCounterValueMutex;
        end;
        2: begin
            ThreadFunction := AddOneToTheCounterValueSpinLock;
            pthread_spin_init(MySpinLock, PTHREAD_PROCESS_PRIVATE);
        end;
    else
        ThreadFunction := AddOneToTheCounterValue;
    end;
end;

```

```

StartTime := Now;
//This function creates the threads and waits on them
for counter := 1 to ThreadCount do
    pthread_create(ThreadIDs[counter], nil, ThreadFunction, nil);

//Let's wait for the threads to finish
for counter := 1 to ThreadCount do
    pthread_join(ThreadIDs[counter], nil);

EndTime := now;

//Let's deallocate any resources required.
case cbSynchMethod.ItemIndex of
    1: begin
        pthread_mutexattr_destroy(MyMutexAttr);
        pthread_mutex_destroy(MyMutex);
    end;
    2: pthread_spin_destroy(MySpinLock);
end;

//Now we can display the result of the thread.
lblResult.Caption := floattostr(ValueToIncrease);

//Now let's calculate the correct value and what it should be.
lblShouldBeValue.Caption := inttostr(IncrementValue * ThreadCount * 1000000);

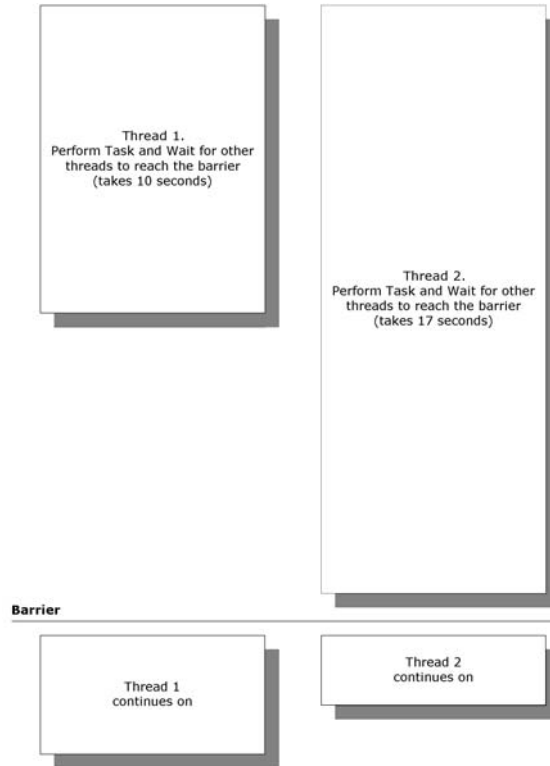
//Display the time taken.
lblTimeTaken.Caption := FormatDateTime('hh:nn:ss', EndTime - StartTime);
end;

end.

```

The last synchronization method we will discuss is barriers. A barrier is a checkpoint that can be used between a number of threads so that no thread proceeds past the barrier unless all the threads have reached the barrier. Barriers are a way of allowing threads to catch up with each other at a predetermined point. Barriers work by having a counter, with each value in the counter representing a single thread that needs to reach the barrier. Whenever a thread reaches the barrier checkpoint, this barrier will decrease the value by one. When the tally finally reaches zero, then all the threads can proceed past the barrier. Listing 6.21 demonstrates using barriers with TThread descendants to show how barriers can be used to allow threads to reach a certain point in the application at a specific time.

Figure 6.11 -
Two threads
using a barrier



Listing 6.21 - Using barriers

```

unit unitDisplayForm;

interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
    Libc, QStdCtrls, IdBaseComponent, IdComponent, IdRawBase, IdRawClient,
    IdIcmpClient, QComCtrls;

type
    TfrmMain = class(TForm)
        Label1: TLabel;
        Label2: TLabel;
        IdIcmpClient1: TIdIcmpClient;
        seDelayAmount: TSpinEdit;
        Label3: TLabel;
        Label4: TLabel;
        edIPAddress: TEdit;
        Button1: TButton;
        mmInfo: TMemo;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    
```



```

end;

TInfoUpdateThread = class(TThread)
protected
    LineToAdd: string;
    procedure AddLine;
    procedure AddInfo(S: string);
end;

TWaitingThread = class(TInfoUpdateThread)
protected
    FDelayAmount: integer;
    procedure Execute; override;
public
    constructor Create(DelayAmountSeconds: integer);
end;

TPingThread = class(TInfoUpdateThread)
protected
    FIPAddress: string;
    procedure Execute; override;
public
    constructor Create(IPAddress: string);
end;

var
    frmMain: TfrmMain;

    //The Barrier Variable
    ApplicationBarrier: pthread_barrier_t;
    ApplicationBarrierAttr: pthread_barrierattr_t;

implementation

{$R *.xfm}

{ TInfoUpdateThread }

procedure TInfoUpdateThread.AddInfo(S: string);
begin
    LineToAdd := S;
    Synchronize(AddLine);
end;

procedure TInfoUpdateThread.AddLine;
begin
    frmMain.mmInfo.Lines.Add(LineToAdd);
end;

{ TWaitingThread }

constructor TWaitingThread.Create(DelayAmountSeconds: integer);
begin
    inherited Create(true);

    FDelayAmount := DelayAmountSeconds;
    Resume;

```

```

end;

procedure TWaitingThread.Execute;
begin
    //This thread will simply wait for 10 seconds and then wait
    //for the other threads to catch up if needed
    AddInfo('Wait Thread: Starting @ '+TimeToStr(Now));

    //Let's wait for the time on the thread
    Sleep(FDelayAmount * 1000);

    AddInfo('Wait Thread: Waiting for the barrier @ '+TimeToStr(Now));
    pthread_barrier_wait(ApplicationBarrier);
    AddInfo('Wait Thread: Everyone is at the barrier @ '+TimeToStr(Now));
end;

{ TPingThread }

constructor TPingThread.Create(IPAddress: string);
begin
    inherited Create(true);

    FIPAddress := IPAddress;
    Resume;
end;

procedure TPingThread.Execute;
var
    IdIcmpClient: TIcmpClient;
    WasSuccessful: boolean;
begin
    AddInfo('Ping Thread: Starting @ '+TimeToStr(Now));
    //Ping the Local IP address
    IdIcmpClient := TIcmpClient.Create(nil);
    try
        IdIcmpClient.Host := FIPAddress;
        try
            IdIcmpClient.Ping;
            WasSuccessful := true;
        except
            WasSuccessful := false;
        end;
    finally
        IdIcmpClient.Free;
    end;

    AddInfo('Ping Thread: Waiting on the barrier @ '+TimeToStr(Now));
    //Now let's wait on the barrier
    pthread_barrier_wait(ApplicationBarrier);
    AddInfo('Ping Thread: Everyone is at the barrier @ '+TimeToStr(Now));
end;

procedure TfrmMain.Button1Click(Sender: TObject);
begin
    //Let's initialize the barrier
    pthread_barrier_init(ApplicationBarrier, ApplicationBarrierAttr, 2);

```

```

mmInfo.Lines.Add('Threads created at '+TimeToStr(Now));
//Now we can create the threads
TWaitingThread.Create(seDelayAmount.Value);
TPingThread.Create(edIPAddress.Text);
end;

end.

```

Thread-Specific Data

In many multi-threaded applications, different threads will want to access a variable that can be accessed by all threads, such as when increasing some global counter. There are times, however, when a thread may want to use a variable that contains different values depending on which thread the data is being accessed by.

In LinuxThreads, whenever a thread is created it will also create a unique storage area for the thread to contain data specific to the thread. The area that is created is known as the thread-specific data area.

```

type
  pthread_key_t = Cardinal;
  TPTHreadKey = pthread_key_t;

```

Storing and reading values from the thread-specific data area is done by a mechanism called keys. A key is really a variable of type `TPTHreadKey` that can be accessed by many threads. Data is not actually read or written from this `TPTHreadKey` variable, but the key is used as an index into a specific value in the thread-specific data area.

To use a key from a thread, the `TPTHreadKey` variable needs to be declared outside the scope of any functions or classes, so that the `TPTHreadKey` is accessible and will live for the lifetime of the application.

A key cannot be used until it is created, however. The key is created by a call to the `pthread_key_create` function.

```

type
  TKeyValueDestructor = procedure(ValueInKey: Pointer); cdecl;

function pthread_key_create(var Key: TPTHreadKey;
  DestrFunction: TKeyValueDestructor): Integer; cdecl;

```

The `pthread_key_create` function takes two parameters. The first parameter is the `TPTHreadKey` variable, which will be used to store thread-specific data. The second parameter is the function that will be called to clean up any memory that may have been allocated to a thread's data for that key. When a thread terminates or is canceled, this destructor function will be called within the context of the terminated thread.

Once the thread key has been created with the `pthread_key_create` function, each thread can now read and write its own specific data to the key using the `pthread_getspecific` and `pthread_setspecific` functions respectively.

```

function pthread_setspecific(Key: TPTHreadKey; Ptr: Pointer): Integer; cdecl;
function pthread_getspecific(Key: TPTHreadKey): Pointer; cdecl;

```

To write a value to the key for the current calling thread, the `pthread_setspecific` function is used. The `pthread_setspecific` function takes two parameters, the first being the `TPTHreadKey` variable and the second a pointer that will be stored for the calling thread

only. The only data that can be stored in a thread-specific key is a pointer. As pointers can point to any type of data or record structure, this means that thread-specific data is fairly flexible.

Reading the value from a thread-specific key is also a simple matter. To read from the key, you would only need to use the `pthread_getspecific` function. This function takes a single parameter, which returns the pointer value from the key for the calling thread. The value that is returned will either be a valid pointer if the thread-specific data has been set already by the current thread or `nil` if the data has not been set on the current thread.

Once you are finished with a thread key, removing it is as simple as calling the `pthread_key_delete` function. This function simply takes a single parameter, which is the `TPThreadKey` variable thread key. Using this function, however, will not call the destructor function for the keys and should be used with caution.

```
function pthread_key_delete(Key: TPThreadKey): Integer; cdecl;
```

So you may be wondering, where do I use thread-specific data? If you have examined the Web server application earlier in the chapter, you will see that the data for each thread is stored by using arrays of values and each thread accesses its own individual array element to obtain the thread's specific piece of data.

But the threaded Web server was a little different. In this example, you knew exactly how many threads were being created in the thread pool, so defining the variables in the application was an easy job. But what about a scenario where you do not know how many threads will be created and you are using the values that were allocated specifically for the thread?

Function libraries, such as a Kylix unit or a shared object, have the most to gain from using thread-specific data. In many cases, old libraries of functions were created under the assumption that the unit would only be used by a single thread and as a result variables were declared in the unit's interface or implementation section. Take, for example, a simple Kylix unit that simply counts how many times a function has been called. The older version of the unit when it was used for a single-threaded application might look like the code in Listing 6.22.

Listing 6.22 - A simple unit that is not designed for multiple threads

```
unit FunctionCallerCounter;

interface

procedure CallSomeProc;
function HowManyTimesCalled: integer;

implementation

var
    __HowManyTimesCalled: integer = 0;

procedure CallSomeProc;
begin
    //Do something

    //increase the number of times this function has been called.
```

```

    inc(__HowManyTimesCalled);
end;

function HowManyTimesCalled: integer;
begin
    Result := __HowManyTimesCalled;
end;

end.

```

As you can see, the unit will increase the same variable, `__HowManyTimesCalled`, regardless of what thread is calling the functions. To make this so that multiple threads can access their own counter, thread keys are used to store the values for their own thread. As a result, we can change the implementation of the functions without changing the interface to the functions. This is handy if we have some legacy Pascal units that you use that need to be multi-threaded. Listing 6.23 demonstrates how the unit in Listing 6.22 can be changed so that the unit is now threadsafe.

Listing 6.23 - A multi-threaded version library using thread keys

```

unit FunctionCallerCounter;

interface

procedure CallSomeProc;
function HowManyTimesCalled: integer;

implementation

uses
    Libc;

var
    HowManyTimesCalledKey: TThreadKey;
    RunOnce: integer = PTHREAD_ONCE_INIT;

procedure RemoveHowManyTimesKeyValue(ValueInKey: Pointer); cdecl;forward;

procedure CallSomeProc;
var
    Value: integer;
    Data: ^Integer;
begin
    //Do something

    //increase the number of times this function has been called.
    Data := pthread_getspecific(HowManyTimesCalledKey);
    if Data = nil then
        Value := 0
    else
        Value := Data^;
    inc(Value);

    //Let's allocate the memory
    //Store the new key value back in
    if Data = nil then

```

```

    new(Data);

    Data^ := Value;
    pthread_setspecific(HowManyTimesCalledKey, Data);
end;

function HowManyTimesCalled: integer;
var
    Data: ^Integer;
begin
    //Return the number of times this function
    //has been called on this thread
    Data := pthread_getspecific(HowManyTimesCalledKey);
    if Data = nil then
    begin
        //No one has called the function yet
        Result := 0;
    end else
        Result := Data^;
    end;
end;

{ Destruction function for the key }

procedure RemoveHowManyTimesKeyValue(ValueInKey: Pointer); cdecl;
begin
    //This function needs to remove the data
    FreeMem(ValueInKey, sizeof(integer));
end;

initialization
    pthread_key_create(HowManyTimesCalledKey, RemoveHowManyTimesKeyValue);
finalization
    pthread_key_delete(HowManyTimesCalledKey);
end.

```

Signals and Threads

In Chapter 5, you were introduced to the concept of signals. Signals allow one process to communicate a message to another process. The message that is sent is normally a message to denote that some event has occurred. There is a predefined list of signals that can be passed, as shown in Table 5.1. The SIGKILL signal may be sent to another process to specify that the process should be terminated or the SIGALRM signal to notify the process of some alarm being received.

Signals within a single-threaded application are difficult enough to use, but when multiple threads are involved, things are even worse. A signal in a single-threaded process would have its own signal mask and signal handlers. When multiple threads are introduced into the equation, things get a little complicated due to the way that threads are created under Linux.

The reason for this is that under Linux each newly created thread is actually a process and will share memory space, file descriptors, the working directory, and signal handlers.

Because each thread is internally a process, the signal mask for a thread can change by use of the `pthread_sigmask` function. This function works on a process in a similar way to

the `sigprocmask` function. This means that if a signal is sent to the thread but not all threads block the signal, the signal will still get through. Also, when a signal is sent to the process only one thread at most will receive the signal. Sending a signal to a process group using the `killpg` function will give the ability to send a signal to all threads.

You should also note that under LinuxThreads, some signals that are sent to the process will result in the process being terminated. Signals such as `SIGKILL` are like this. When one thread in a process receives this type of signal, all of the threads will be terminated. Other signals that do not result in the thread being terminated, such as the `SIGALRM` signal, will actually be sent to the thread that caused the signal to be sent. For example, using the `alarm` function from one thread will result in only that thread being sent the `SIGALRM` signal.

Another point of note when working with signals and POSIX threads is that some versions of LinuxThreads use the `SIGUSR1` and `SIGUSR2` signals to manage the suspension of threads from a process. As a result, it is not a good idea to use these threads in your multi-threaded applications.

Mixing signals and threads is possible, but doing this under Linux is not recommended. If you need notification of events, it is often best to use better synchronization methods, such as mutexes, semaphores, barriers, or conditional variables. If you can avoid the use of signals in the multi-threaded applications, your application will be all the better for it.

Integrating with CLX's TThread Object

Many of the functions that are used to manage POSIX threads within Kylix rely on the thread identifier that is returned after a successful call to the `pthread_create` function. Working with the POSIX thread is then a matter of using this thread identifier and making the appropriate function calls to obtain the desired effect.

But as was mentioned before, creating multi-threaded applications is difficult and adds much complexity into your application. If you want to reduce the complexity of your multi-threaded application, use the CLX class `TThread` which wraps much of the functionality of POSIX threads and is simple to use compared with calling POSIX thread functions directly. The thread support in `TThread` does not have all the features that the POSIX threads have, so in some cases you may want to mix some code between POSIX calls and the `TThread` class.

The `ThreadID` property of the `TThread` class is what is used to obtain the thread identifier that is passed to any POSIX thread function that requires a thread identifier. This can be seen in Listing 6.24 in the `SendSignal` function method of `TPOSIXThread`.

Some POSIX functions work on the calling thread only, such as the `pthread_sigmask` and the `pthread_kill_other_threads_np` functions demonstrated in Listing 6.24's `SigMask` and `KillTheOtherThreads` methods. Rather than obtain the `ThreadID` of the thread, the call should be made only within the context of the thread. In this example, the code in which the POSIX function calls must be made within the current thread are placed in the protected section so that they can only be called by that class and any descendant. This should enforce that the `SigMask` and `KillTheOtherThreads` functions are only called within the context of the calling thread.

Listing 6.24 - Extending the behavior of TThread

```

unit POSIXThread;

(*
    This class demonstrates how we can extend the normal
    TThread class by using some of the POSIX threading functions.
*)

interface

uses
    Classes, Libc;

type
    TPOSIXThread = class(TThread)
    protected
        //Killing all the other threads
        procedure KillTheOtherThreads;
        //Methods for Handling the Signal Mask for the current thread
        function SigMask(How: Integer; const NewMask: PSigSet; OldMask: PSigSet): integer;
    public
        //Send a signal to this thread
        function SendSignal(SigNum: integer): integer;
    end;

implementation

{ TPOSIXThread }

procedure TPOSIXThread.KillTheOtherThreads;
begin
    //Kill all the threads in the application
    //except the calling thread.
    pthread_kill_other_threads_np;
end;

function TPOSIXThread.SendSignal(SigNum: integer): integer;
begin
    //This function will send a signal to the thread.
    //This uses the Handle property of TThread to obtain the thread
    //identifier so that it can be used with the POSIX Thread functions.
    Result := pthread_kill(ThreadID, SigNum);
end;

function TPOSIXThread.SigMask(How: Integer; const NewMask: PSigSet;
    OldMask: PSigSet): integer;
begin
    //This function simply wraps it for the calling thread.
    Result := pthread_sigmask(How, NewMask, OldMask);
end;

end.

```

Much of the support you need for threads within your application is contained within the TThread class. While looking for ways to extend the TThread object with POSIX threading features, most of the pthread functions were already there. When you do want to mix

TThread and POSIX threading functions within your application, it may be to add features such as POSIX's various synchronization methods such as semaphores, mutexes, condition variables, and others.

API Reference

`__pthread_initialize` *LibC.pas*

Syntax

```
procedure __pthread_initialize;
```

Description

The `__pthread_initialize` function will initialize the LinuxThreads library. Normally you will not call this function, because when you create a thread for the first time, the library will initialize itself.

See Also

`pthread_create`

`_pthread_cleanup_pop` *LibC.pas*

Syntax

```
procedure _pthread_cleanup_pop(  
  var Buffer: TPthreadCleanupBuffer;  
  Execute: Integer  
);
```

Description

Each thread can contain a list of functions that are called when a thread is canceled or exits. The `_pthread_cleanup_pop` function will remove the topmost thread cleanup handler routine from the list of cleanup handlers stored in the Buffer parameter.

This function also gives the opportunity to execute the topmost cleanup handler before it removes it from the top of the list.

Parameters

Buffer: The Buffer parameter is a TPthreadCleanupBuffer variable, which is actually a linked list that contains the list of cleanup functions for the current thread. When the thread cancels or exits, the function stored in this list will be executed. This parameter is the buffer used by the current thread and will have its topmost element removed from the list.

Execute: This parameter determines if the cleanup handler should be executed before it is removed from the top of the list. Setting this parameter to a non-zero value will result in the cleanup function being executed. Setting this parameter to 0 will result in the function not being executed.

See Also

`_pthread_cleanup_push`, `_pthread_cleanup_push_defer`, `_pthread_cleanup_pop_restore`

`_pthread_cleanup_pop_restore` LibC.pas**Syntax**

```
procedure _pthread_cleanup_pop_restore(
  Buffer: PPthreadCleanupBuffer;
  Execute: Integer
);
```

Description

The `_pthread_cleanup_pop_restore` function behaves the same way the `_pthread_cleanup_pop` function does in that it will remove the top cleanup handler from the cleanup list. This function is used, however, to restore the cancellation state of the current thread that was previously set to deferred cancellation as a result of a call to the `_pthread_cleanup_push_defer` procedure.

Parameters

Buffer: The Buffer parameter is a `TPthreadCleanupBuffer` variable, which is actually a linked list that contains the list of cleanup functions for the current thread. When the thread cancels or exits, the function stored in this list will be executed. This parameter is the buffer used by the current thread and will have its topmost element removed from the list.

Execute: This parameter determines if the cleanup handler should be executed before it is removed from the top of the list. Setting this parameter to a non-zero value will result in the cleanup function being executed. Setting this parameter to 0 will result in the function not being executed.

See Also

`_pthread_cleanup_push`, `_pthread_cleanup_push_defer`, `_pthread_cleanup_pop`

`_pthread_cleanup_push` LibC.pas**Syntax**

```
procedure _pthread_cleanup_push(
  var Buffer: TPthreadCleanupBuffer;
  Routine: TPthreadCleanupRoutine;
  Arg: Pointer
);
```

Description

Each thread can contain a list of functions that are called when a thread is canceled or exits. The `_pthread_cleanup_push` function will add to the list of functions that are called when the thread terminates. If this function is not called within a thread, no cleanup will occur.

However, whenever this function is called, the cleanup function presented in the linked list `TPthreadCleanupBuffer` variable will be executed on a first-in/last-out basis.

Parameters

Buffer: The Buffer parameter is a `TPthreadCleanupBuffer` variable, which is actually a linked list that contains the list of cleanup functions for the current thread. When the thread cancels or exits, the function stored in this list will be executed.

Routine: This parameter is the cleanup function that will be added to the cleanup list of functions for the current thread.

type

```
TPThreadCleanupRoutine = procedure(p1: Pointer); cdecl;
```

Arg: The Arg parameter is a pointer parameter that will be passed to the Routine parameter when the function is executed.

See Also

`_pthread_cleanup_push_defer`, `_pthread_cleanup_pop`, `_pthread_cleanup_pop_restore`

`_pthread_cleanup_push_defer` LibC.pas

Syntax

```
procedure _pthread_cleanup_push_defer(  
  Buffer: TPthreadCleanupBuffer;  
  Routine: TPthreadCleanupRoutine;  
  Arg: Pointer  
);
```

Description

The `_pthread_cleanup_push_defer` function is similar to the `pthread_cleanup_push` function in that it will add a cleanup handler for the current thread. The only difference is that the `_pthread_cleanup_push_defer` function will cause the thread to set its cancellation mode to deferred and will store the current cancellation state. The cancellation state and current cleanup handlers can then be used with a call to the `pthread_cleanup_pop_restore` function.

Parameters

Buffer: The Buffer parameter is a `TPthreadCleanupBuffer` variable, which is actually a linked list that contains the list of cleanup functions for the current thread. When the thread cancels or exits, the function stored in this list will be executed.

Routine: This parameter is the cleanup function that will be added to the cleanup list of functions for the current thread.

type

```
TPThreadCleanupRoutine = procedure(p1: Pointer); cdecl;
```

Arg: The Arg parameter is a pointer parameter that will be passed to the Routine parameter when the function is executed.

See Also

`_pthread_cleanup_push`, `_pthread_cleanup_pop`, `_pthread_cleanup_pop_restore`

clone LibC.pas**Syntax**

```
function clone(
  fn: TCloneProc;
  ChildStack: Pointer;
  Flags: Integer;
  Arg: Pointer
): Integer;
```

Description

The clone function is the function that allows the ability to run threaded applications. This function behaves like the fork function so that a new process is created. Instead of copying the process in its entirety, the function shares the same memory, file descriptors, and signal handlers. When the new process is created, the function passed in the fn parameter is executed, and it is this functionality that allows the developer to create multiple threads in an application.

Parameters

fn: The fn parameter is a TCloneProc function that is the function that will be executed when the new process is created. The integer returned by this function is the exit code of the new process.

```
type
  TCloneProc = function(Arg: Pointer): Integer; cdecl;
```

ChildStack: The ChildStack parameter is the stack for the new process. The stack cannot be shared between processes, so a new stack needs to be created for the process.

Flags: The Flags parameter determines which parts of the process will be shared between processes. Including the CLONE_VM option means that the new process will share the same memory. Including the CLONE_FS option means that the file system information such as the umask and the current directory will be shared for both processes. Including CLONE_FILES means that file descriptors will be shared between processes. Including CLONE_SIGHAND means that the new process will have the same signal handlers and signal mask in the new process; otherwise, new signal handlers will have to be used. Including the CLONE_PID option means that the new process will share the process ID (pid) of the parent process; otherwise, a new process ID will be created.

Arg: The Arg parameter is a pointer that will be passed to the function in the fn parameter when the new process is created.

Return Value

When a new process can be created, the function will return a positive value that represents the process ID (pid) of the newly created process. If the new process cannot be created, the

function returns `-1` and a call to the `errno` function will either return `EAGAIN`, if there are too many processes on the operating system, or `ENOMEM`, if there is insufficient memory to create the new process.

See Also

`fork`, `pthread_create`

GetCurrentThreadID *LibC.pas*

Syntax

```
function GetCurrentThreadID: TThreadID
```

Description

The `GetCurrentThreadID` is an alias to the `pthread_self` function, which returns the thread handle that represents the current thread.

Return Value

The function returns the thread handle of the current calling thread.

See Also

`pthread_self`, `pthread_equal`

Example

Listing 6.25 - Using the GetCurrentThreadID function

```
procedure TfrmDemo.btnTestThreadIDsClick(Sender: TObject);
var
  CurrentThread, SelfThread: Cardinal;
begin
  CurrentThread := GetCurrentThreadID;
  SelfThread := pthread_self;
  mmInfo.Lines.Add('When Called with "GetCurrentThreadID" the Thread ID is '+
    inttostr(GetCurrentThreadID));
  mmInfo.Lines.Add('When Called with "pthread_self" the Thread ID is '+
    inttostr(pthread_self));
  if pthread_equal(CurrentThread, SelfThread) = 0 then
    mmInfo.Lines.Add('These threads are equal.')
  else
    mmInfo.Lines.Add('These threads are not equal.');
```

pthread_atfork *LibC.pas*

Syntax

```
function pthread_atfork(
  Prepare: TForkHandler;
  Parent: TForkHandler;
  Child: TForkHandler
): Integer;
```

Description

The `pthread_atfork` function is used to specify what functions to perform when the current process calls the fork function. This function is used to specify what procedures to call in the parent and child processes. The procedure defined by `Prepare` is called before the process is copied, the `Parent` procedure is called in the parent process after the process has been copied, and the `Child` procedure is called in the child process after the process has been copied.

The `pthread_atfork` function is used often to make sure that allocations of resources and synchronization mechanisms are in place. This is due to the fact that the fork function makes a copy of the entire process, but only the thread that calls the fork function will be alive in the new process. This also gives the new child process the ability to initialize any new threads that need to be created.

This function can be called multiple times to install multiple handlers when the process forks. Procedures passed into the `Prepare` parameter will be called in the reverse order in which they were added. Procedures passed into the `Parent` and `Child` parameters will be called in the same order in which they were added.

Parameters

Prepare: The `Prepare` parameter is a procedure with no parameters using the cdecl calling convention. This function will be called in the parent process just before the process forks. Alternatively, this handler can be `nil`, in which case no `atfork` `Prepare` handler will be added.

Parent: The `Parent` parameter, like the `Prepare` parameter, is a procedure with no parameters using the cdecl calling convention. This function will be called in the parent process after the process forks and before the fork function returns. This handler can also be `nil`, in which case no `atfork` `Parent` handler will be added.

Child: The `Child` parameter is another procedure with no parameters using the cdecl calling convention. This function will be called in the child process after the process forks and before the fork function returns. This value can also be `nil`, in which case no `atfork` `Child` handler will be added.

Return Value

If the function is successful, the function will return 0. If the function is unsuccessful, which will only be due to insufficient memory, the function will return the error code `ENOMEM`.

See Also

`fork`

Example

Listing 6.26 - Using the `pthread_atfork` function

```
program pthread_atfork_Demo;

{$APPTYPE CONSOLE}
```

```

uses
  Libc, SysUtils;

function DisplayTheCurrentTimeEverySecond(Data: Pointer): integer; cdecl;
begin
  repeat
    writeln('Process ID (', getpid, '): ', TimeToStr(Now));

    sleep(1000);
  until false;
end;

var
  ThreadID: Cardinal;

procedure WhatToDoWithTheChildProcess; cdecl;
begin
  writeln('The process has been forked. This is coming from the new child process.');
```

```

end;

procedure WhatToDoWithTheOriginalProcess; cdecl;
begin
  writeln('The process is just about to be forked.');
```

```

end;

begin
  //Let's install the new child handler
  pthread_atfork(WhatToDoWithTheOriginalProcess, nil, WhatToDoWithTheChildProcess);

  //Let's create the new thread
  pthread_create(ThreadID, nil, DisplayTheCurrentTimeEverySecond, nil);

  writeln('A simple thread application.');
```

```

  writeln('Let's wait for 5 seconds.');
```

```

  sleep(5000);

  //Let's close all forked processes
  if fork 0 then
  begin
    sleep(5000);
  end;
end.
```

pthread_attr_destroy LibC.pas

Syntax

```

function pthread_attr_destroy(
  var Attr: TThreadAttr
):Integer;
```

Description

In the POSIX threads standard, the `pthread_attr_destroy` function will release any resources created when a POSIX thread attribute was originally created using the `pthread_attr`

`_init` function. On Linux, this function does nothing, but it is still a good idea to call it anyway, in case the implementation of LinuxThreads changes.

Parameters

Attr: The Attr parameter is the TThreadAttr structure that was used to initialize the thread attribute in the first place.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_attr_init`

`pthread_attr_getdetachstate` LibC.pas

Syntax

```
function pthread_attr_getdetachstate(
  const Attr: TThreadAttr;
  var DetachState: Integer
):Integer;
```

Description

The `pthread_attr_getdetachstate` function returns the current value of the thread attribute's detach property. This detach property is used to specify whether the thread is joinable by a call to the `pthread_join` function.

Parameters

Attr: The Attr parameter is a valid TThreadAttr thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

DetachState: This is an integer variable that will contain the value that says if the thread attribute is detached or joinable. If the thread attribute can be joined, the parameter will contain the constant `PTHREAD_CREATE_JOINABLE`; otherwise, the value `PTHREAD_CREATE_DETACHED` will be returned.

```
const
  PTHREAD_CREATE_JOINABLE = 0;
  PTHREAD_CREATE_DETACHED = 1;
```

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_detach`, `pthread_attr_setdetachstate`, `pthread_join`

pthread_attr_getguardsize LibC.pas***Syntax***

```
function pthread_attr_getguardsize(
  const Attr: TThreadAttr;
  var Guardsize: LongWord
):Integer;
```

Description

The `pthread_attr_getguardsize` function is used to obtain the guard size value from a `TThreadAttr` structure. When applied to a thread, the guard size is used to protect the thread from having a stack overflow which causes the corruption of data in other threads.

Parameters

Attr: The `Attr` parameter is a valid `TThreadAttr` thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

Guardsize: When the function returns, this parameter will hold the guard size specified in the thread attribute.

Return Value

If the function is successful, the function will return 0 and store the guard size defined in the `Guardsize` parameter. If the function is unsuccessful, however, the function will return a valid error code.

See Also

`pthread_attr_getstack`, `pthread_attr_getstackaddr`, `pthread_attr_getstacksize`,
`pthread_attr_setguardsize`, `pthread_attr_setstack`, `pthread_attr_setstackaddr`,
`pthread_attr_setstacksize`

pthread_attr_getinheritsched LibC.pas***Syntax***

```
function pthread_attr_getinheritsched(
  const Attr: TThreadAttr;
  var Inherit: Integer
):Integer;
```

Description

The `pthread_attr_getinheritsched` function is used to obtain the `__inheritsched` field from a `TThreadAttr` structure. The inherited schedule field in a thread attribute defines if a newly created POSIX thread will inherit the same scheduling characteristics as the thread that created it.

Parameters

Attr: The Attr parameter is a valid TThreadAttr thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

Inherit: When this function is successful, the Inherit property will show if the thread attribute has the inherit schedule flag set. If the thread attribute does inherit the schedule from the parent thread, this value will be PTHREAD_INHERIT_SCHED. Otherwise, the thread attribute will be used to create a thread with specific scheduling qualities and this parameter will contain a value of PTHREAD_EXPLICIT_SCHED.

```
const
    PTHREAD_INHERIT_SCHED = 0;
    PTHREAD_EXPLICIT_SCHED = 1;
```

Return Value

If the function is successful, the function will return 0, and the Inherit field will show if the thread attribute inherit flag is set. If the function is unsuccessful, however, the function will return a valid error code.

See Also

pthread_attr_setinheritsched, pthread_attr_getschedparam, pthread_attr_getschedpolicy, pthread_attr_getscope, pthread_attr_setschedparam, pthread_attr_setschedpolicy, pthread_attr_setscope

pthread_attr_getschedparam LibC.pas

Syntax

```
function pthread_attr_getschedparam(
    const Attr: TThreadAttr;
    var Param: TSchedParam
):Integer;
```

Description

The pthread_attr_getschedparam function is used to read the scheduling parameter (__schedparam field) from a TThreadAttr structure. The value read is used for setting the priority of the thread.

Parameters

Attr: The Attr parameter is a valid TThreadAttr thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

Param: When the function call is successful, the function stores the scheduling parameter, the priority for the thread attribute in this parameter.

Return Value

If the function is successful, it will return 0 and the scheduling parameter will be stored in the Param parameter. If the function is unsuccessful, however, it will return a valid error code.

See Also

pthread_attr_getinheritsched, pthread_attr_getschedpolicy, pthread_attr_getscope,
pthread_attr_setinheritsched, pthread_attr_setschedparam, pthread_attr_setschedpolicy,
pthread_attr_setscope

pthread_attr_getschedpolicy LibC.pas*Syntax*

```
function pthread_attr_getschedpolicy(  
  const Attr: TThreadAttr;  
  var Policy: Integer  
):Integer;
```

Description

The pthread_attr_getschedpolicy function returns the scheduling policy that is stored within a thread attribute. The scheduling policy is used when creating a thread to define how scheduling will occur between the threads. The possible values returned from this function will be round robin scheduling, scheduling on a first-in/first-out basis, and explicit scheduling.

Parameters

Attr: The Attr parameter is a valid TThreadAttr thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

Policy: After a successful function call, this parameter will hold the scheduling policy of the thread attribute. This value will either be SCHED_FIFO, SCHED_RR, or SCHED_OTHER. Having SCHED_FIFO returned in this parameter signifies that scheduling for this thread will be done on a first-in/first-out basis. Having SCHED_RR returned in this parameter means that the scheduling for the thread will be done using a round robin algorithm, and having SCHED_OTHER returned means that the scheduling value will be calculated using the value specified by a call to the pthread_attr_setschedparam function.

```
const  
  SCHED_OTHER    = 0;  
  SCHED_FIFO     = 1;  
  SCHED_RR       = 2;
```

Return Value

If the function is successful, it will return 0 and the scheduling policy will be placed in the Policy parameter. If the function is unsuccessful, however, it will return a valid error code.

See Also

pthread_attr_getinheritsched, pthread_attr_getschedparam, pthread_attr_getscope,
pthread_attr_setinheritsched, pthread_attr_setschedparam, pthread_attr_setschedpolicy,
pthread_attr_setscope

pthread_attr_getscope LibC.pas***Syntax***

```
function pthread_attr_getscope(
  const Attr: TThreadAttr;
  var Scope: Integer
):Integer;
```

Description

The POSIX standard states that the `pthread_attr_getscope` function returns whether or not scheduling for a thread created using the thread attribute passed into the `Attr` parameter will be calculated system wide or within the process. Unfortunately, only system-wide scheduling can be performed with Linux's implementation of POSIX threads, so although this function is declared in `LibC.pas`, it is essentially useless.

Parameters

`Attr`: The `Attr` parameter is a valid `TThreadAttr` thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

`Scope`: When the function returns, this parameter will hold the status of the scoping field of the thread attribute, which will either be `PTHREAD_SCOPE_SYSTEM` for system-wide scheduling or `PTHREAD_SCOPE_PROCESS` for process-wide scheduling.

```
const
  PTHREAD_SCOPE_SYSTEM = 0;
  PTHREAD_SCOPE_PROCESS = 1;
```

Return Value

If the function is successful, it will return 0 and the scheduling scope will be placed in the `Scope` parameter. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_attr_getinheritsched`, `pthread_attr_getschedparam`, `pthread_attr_getschedpolicy`, `pthread_attr_setinheritsched`, `pthread_attr_setschedparam`, `pthread_attr_setschedpolicy`, `pthread_attr_setscope`

pthread_attr_getstack LibC.pas***Syntax***

```
function pthread_attr_getstack(
  const Attr: TThreadAttr;
  var StackAddr: Pointer;
  var StackSize: size_t
):Integer;
```

Description

The `pthread_attr_getstack` function is a combination of the `pthread_attr_getstackaddr` and the `pthread_attr_getstacksize` functions. This function returns the address of the thread's

stack in memory and the size of the stack. This function can only be called after the thread has been created with the `pthread_create` function.

Parameters

Attr: The `Attr` parameter is a valid `TThreadAttr` thread attribute structure, which is used to specify the attributes of a POSIX thread when it is created.

StackAddr: After a successful function call, this parameter will hold the pointer to where the thread's stack is located.

StackSize: After a successful function call, this parameter will hold the size of the thread.

Return Value

If the function is successful, it will return 0, the thread's stack address will be returned to the `StackAddr` parameter, and the size of the thread's stack will be returned in the `StackSize` parameter. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_attr_getstackaddr`, `pthread_attr_getstacksize`, `pthread_attr_setstack`,
`pthread_attr_setstackaddr`, `pthread_attr_setstacksize`

***pthread_attr_getstackaddr* LibC.pas**

Syntax

```
function pthread_attr_getstackaddr(
  const Attr: TThreadAttr;
  var StackAddr: Pointer
):Integer;
```

Description

The `pthread_attr_getstackaddr` function returns the location of the stack for a thread attribute that was previously stored in a thread attribute using either `pthread_attr_setstack` or `pthread_attr_setstackaddr`.

Parameters

Attr: This is the thread attribute that contains the location of the stack.

StackAddr: This is a pointer variable that will contain the location of the thread's stack when the function returns.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, however, it will return a valid error code.

This function will always return the value 0 and store the location of the stack in the `StackAddr` parameter if it was set, or nil will be stored in the `StackAddr` parameter if it was not set.

See Also

`pthread_attr_setstack`, `pthread_attr_setstackaddr`, `pthread_attr_setstacksize`, `pthread_attr_getstack`, `pthread_attr_getstacksize`, `pthread_attr_getguardsize`, `pthread_attr_setguardsize`

pthread_attr_getstacksize LibC.pas*Syntax*

```
function pthread_attr_getstacksize(
  const Attr: TThreadAttr;
  var StackSize: LongWord
):Integer;
```

Description

The `pthread_attr_getstacksize` function returns the size in bytes of a thread attribute's stack that was originally set using `pthread_attr_setstack` or `pthread_attr_setstacksize`.

Parameters

Attr: This is the thread attribute that contains the stack size information.

StackSize: After the function returns the size of the stack stored within the thread, the attribute `Attr` will be contained in this parameter.

Return Value

This function always returns 0 and will return the stack size of the thread in the `StackSize` parameter.

See Also

`pthread_attr_setstack`, `pthread_attr_setstackaddr`, `pthread_attr_setstacksize`, `pthread_attr_getstack`, `pthread_attr_getstackaddr`, `pthread_attr_getguardsize`, `pthread_attr_setguardsize`

pthread_attr_init LibC.pas*Syntax*

```
function pthread_attr_init(
  var Attr: TThreadAttr
):Integer;
```

Description

The `pthread_attr_init` function initializes a thread attribute with the default values. The default values for thread attributes are that a new thread created with the attribute passed into the `Attr` parameter will be joinable and scheduling parameters for the thread will be set explicitly.

Parameters

Attr: This parameter is the thread attribute that will have its fields set to the default values.

Return Value

On Linux, this function will always return 0. After the call to this function, the thread attribute passed into the Attr parameter will be set.

See Also

pthread_create, pthread_attr_destroy

Example

See Listing 6.3 - Creating a thread attribute.

pthread_attr_setdetachstate LibC.pas**Syntax**

```
function pthread_attr_setdetachstate(
  var Attr: TThreadAttr;
  DetachState: Integer
):Integer;
```

Description

The pthread_attr_setdetachstate function defines if a thread cannot be joined by other threads using the pthread_join function. POSIX threads are normally detached by default unless specified otherwise.

Parameters

Attr: This is the thread attribute that will have its detach state field changed.

DetachState: This parameter defines if the POSIX thread created with this thread attribute will be joinable or detached. If the thread is to be joinable, this value will be PTHREAD_CREATE_JOINABLE. If the thread will be detached, then this value should be PTHREAD_CREATE_DETACHED.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

pthread_attr_getdetachstate

Example

See Listing 6.4 - Using thread attributes.

pthread_attr_setguardsize LibC.pas**Syntax**

```
function pthread_attr_setguardsize(
  var Attr: TThreadAttr;
```

```
Guardsize: LongWord
):Integer;
```

Description

The `pthread_attr_setguardsize` function defines the guard size of the thread. When applied to a thread, the guard size is used to protect the thread from having a stack overflow, which causes the corruption of data in other threads. Normally this function is used when large amounts of data may be placed on the stack. By altering the guard size, you can obtain more control over the memory usage of each thread.

Parameters

Attr: This is a thread attribute variable that will have the information about the guard size changed.

Guardsize: This is the new guard size of the thread. **Guardsize** should be less than the size of the stack.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_attr_getguardsize`, `pthread_attr_getstack`, `pthread_attr_getstackaddr`, `pthread_attr_getstacksize`, `pthread_attr_setstack`, `pthread_attr_setstackaddr`, `pthread_attr_setstacksize`

***pthread_attr_setinheritsched* LibC.pas**

Syntax

```
function pthread_attr_setinheritsched(
  var Attr: TThreadAttr;
  Inherit: Integer
):Integer;
```

Description

The `pthread_attr_setinheritsched` function defines if a thread will inherit the scheduling attributes from the thread that creates the new thread.

Parameters

Attr: This is the `TThreadAttr` variable that will have its `inherit` flag altered.

Inherit: This parameter defines if the newly created thread will inherit the scheduling properties of the thread that created it or if the thread will define its own scheduling information. This value will be either the constant value `PTHREAD_INHERIT_SCHED`, which means the thread will inherit the schedule, or `PTHREAD_EXPLICIT_SCHED`, which means that the thread will define its own scheduling properties.

```
const
  PTHREAD_INHERIT_SCHED = 0;
```



```
PTHREAD_EXPLICIT_SCHED = 1;
```

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_attr_getinheritsched`, `pthread_attr_getschedparam`, `pthread_attr_getschedpolicy`,
`pthread_attr_getscope`, `pthread_attr_setschedparam`, `pthread_attr_setschedpolicy`,
`pthread_attr_setscope`

Example

See Listing 6.4 - Using thread attributes.

pthread_attr_setschedparam LibC.pas

Syntax

```
function pthread_attr_setschedparam(  
  var Attr: TThreadAttr;  
  Param: PSchedParam  
):Integer;
```

Description

The `pthread_attr_setschedparam` function is used to specify the scheduling priority of a thread created with the thread attribute passed into the `Attr` parameter. Setting the scheduling priority is only useful when the scheduling policy is using real-time scheduling, such as the round robin or first-in/first-out scheduling policy.

Parameters

Attr: This is the thread attribute that will have its priority information changed.

Param: The `Param` function is a pointer to a `TSchedParam` structure that contains the new priority value. The value in the `TSchedParam`'s `sched_priority` field must be between the minimum and maximum values for the scheduling policy that is used. The minimum and maximum values for the scheduling policies can be found by using the `sched_get_priority_min` and `sched_get_priority_max` functions.

```
type  
  TSchedParam = record  
    sched_priority: Integer;  
  end;
```

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code. The function will only result in an error if the new scheduling priority is invalid for the thread attribute's scheduling policy.

See Also

pthread_attr_getinheritsched, pthread_attr_getschedpolicy, pthread_attr_getscope,
pthread_attr_setinheritsched, pthread_attr_getschedparam, pthread_attr_setschedpolicy,
pthread_attr_setscope

Example

See Listing 6.4 - Using thread attributes.

pthread_attr_setschedpolicy LibC.pas

Syntax

```
function pthread_attr_setschedpolicy(
  var Attr: TThreadAttr;
  Policy: Integer
):Integer;
```

Description

The pthread_attr_setschedpolicy function sets the scheduling policy for the thread attribute Attr. The scheduling policy is used to define CPU usage allocated to threads. The possible values for the scheduling policy are round robin scheduling, scheduling on a first-in/first-out (FIFO) basis, and explicit scheduling. Round robin and FIFO scheduling are only available to processes running as the superuser as these policies are for real-time systems only.

Parameters

Attr: This is the thread attribute that will have its scheduling policy information changed.

Policy: This is the scheduling policy that will be set for the thread attribute. The available options for this parameter are SCHED_RR for round robin scheduling, SCHED_FIFO for first-in/first-out scheduling, and SCHED_OTHER for explicit scheduling. Any value other than these three will result in the function failing and returning an EINVAL error code.

```
const
  SCHED_OTHER    = 0;
  SCHED_FIFO     = 1;
  SCHED_RR       = 2;
```

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, which will be as a result of an incorrect value being passed to the Policy parameter, it will return a valid error code.

See Also

pthread_attr_getinheritsched, pthread_attr_getschedparam, pthread_attr_getscope,
pthread_attr_setinheritsched, pthread_attr_setschedparam, pthread_attr_getschedpolicy,
pthread_attr_setscope

Example

See Listing 6.4 - Using thread attributes.

pthread_attr_setscope LibC.pas*Syntax*

```
function pthread_attr_setscope(
  var Attr: TThreadAttr;
  Scope: Integer
):Integer;
```

Description

In the POSIX specification, the `pthread_attr_setscope` function is used to specify if the scheduling for the thread is to be done within the current process or system wide. Unfortunately, the Linux implementation of POSIX threads only supports system-wide scheduling, so this function has no effect in Linux.

Parameters

Attr: This is the thread attribute that will have its scope information changed.

Scope: This is the scope to specify. This will be one of the following constants: `PTHREAD_SCOPE_PROCESS` to specify scheduling to be performed within the current process or `PTHREAD_SCOPE_SYSTEM` to state that the scheduling will be done system wide. Only `PTHREAD_SCOPE_SYSTEM` is supported on Linux, as scheduling is performed by the Linux kernel.

```
const
  PTHREAD_SCOPE_SYSTEM = 0;
  PTHREAD_SCOPE_PROCESS = 1;
```

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_attr_getinheritsched`, `pthread_attr_getschedparam`, `pthread_attr_getschedpolicy`, `pthread_attr_setinheritsched`, `pthread_attr_setschedparam`, `pthread_attr_setschedpolicy`, `pthread_attr_getscope`

pthread_attr_setstack LibC.pas*Syntax*

```
function pthread_attr_setstack(
  var Attr: TThreadAttr;
  StackAddr: Pointer;
  StackSize: size_t
):Integer;
```

Description

The `pthread_attr_setstack` function will set the location in memory of the stack and the size of the stack in bytes, so whenever a thread is created using this thread attribute, it will have these settings for a thread. The stack of a thread is used to store local variables and parameters when calling functions within the thread. It is very unlikely that you will need to change the size of the stack unless you are allocating large amounts of memory within the local variables, using recursive functions, or do not require a large stack for the thread.

If you have not set the stack information using one of the functions that begin with `pthread_attr_setstack`, the stack will be set to the default size for a thread.

Parameters

Attr: This is the thread attribute which will have its stack information altered.

StackAddr: The `StackAddr` is a pointer to a block of memory that will contain the stack.

StackSize: This parameter is the size of the stack in bytes. This parameter needs to be within the allowable stack size for a thread. The minimum stack size can be found by a call to `sysconf(_SC_THREAD_STACK_MIN)` and the maximum value can be found by a call to `sysconf(_SC_THREAD_STACK_MAX)`.

Return Value

If the function is successful, it will return 0. The function is likely to be unsuccessful if the stack size is not valid, and this will result in the function returning `EINVAL`.

See Also

`pthread_attr_setstackaddr`, `pthread_attr_setstacksize`, `pthread_attr_getstack`,
`pthread_attr_getstackaddr`, `pthread_attr_getstacksize`, `pthread_attr_getguardsize`,
`pthread_attr_setguardsize`

`pthread_attr_setstackaddr` LibC.pas

Syntax

```
function pthread_attr_setstackaddr(
  var Attr: TThreadAttr;
  StackAddr: Pointer
):Integer;
```

Description

The `pthread_attr_setstackaddr` function stores within a thread attribute the location of a memory area to be used as the stack by any thread created with the attribute `Attr`. This function should be used in conjunction with `pthread_attr_setstacksize` to properly define the stack for a new thread.

Parameters

Attr: This is the thread attribute that will store the location of a new thread's stack.

StackAddr: The StackAddr is a pointer to a memory block that will become the thread's stack when the thread is created with the thread attribute Attr.

Return Value

This function will store the location used for the stack address and always return 0.

See Also

pthread_attr_setstack, pthread_attr_setstacksize, pthread_attr_getstack, pthread_attr_getstackaddr, pthread_attr_getstacksize, pthread_attr_getguardsize, pthread_attr_setguardsize

pthread_attr_setstacksize LibC.pas

Syntax

```
function pthread_attr_setstacksize(
  var Attr: TThreadAttr;
  StackSize: LongWord
):Integer;
```

Description

The pthread_attr_setstacksize function changes the value of the stack size in a thread attribute Attr so that threads created with this thread attribute will have a stack of a particular size. This function needs to be used in conjunction with pthread_attr_setstackaddr to fully define the stack for a thread.

Parameters

Attr: This is the thread attribute that will store the location of a new thread's stack.

StackSize: This parameter is the size of the stack in bytes. This parameter needs to be within the allowable stack size for a thread. The minimum stack size can be found by a call to sysconf(_SC_THREAD_STACK_MIN) and the maximum value can be found by a call to sysconf(_SC_THREAD_STACK_MAX).

Return Value

If the function is successful, it will return 0. The function will return the error code EINVAL if the stack size is not within the minimum and maximum size for a thread.

See Also

pthread_attr_setstack, pthread_attr_setstackaddr, pthread_attr_getstack, pthread_attr_getstackaddr, pthread_attr_getstacksize, pthread_attr_getguardsize, pthread_attr_setguardsize

pthread_barrier_destroy LibC.pas

Syntax

```
function pthread_barrier_destroy(
  var Barrier: TPthreadBarrier
):Integer;
```

Description

The `pthread_barrier_destroy` function releases any resources allocated to a barrier by a call to `pthread_barrier_init`. Before a barrier is destroyed, it needs to make sure it is not still in use. If the barrier is still in use, it cannot be destroyed.

Parameters

Barrier: This is the barrier variable to destroy.

Return Value

If the function is successful, it will return 0. The function will only fail if the barrier is still in use by other threads. If this is the case, the function will return the EBUSY error code.

See Also

`pthread_barrier_init`, `pthread_barrier_wait`

`pthread_barrier_init` LibC.pas

Syntax

```
function pthread_barrier_init(
  var Barrier: TPtrthreadBarrier;
  const Attr: TPtrthreadBarrierAttribute;
  Count: Cardinal
):Integer;
```

Description

A barrier is a synchronization mechanism that allows a group of threads to stop at certain places and wait for all other threads to reach that same point. This function creates the resources required to set up a POSIX barrier and sets up the number of threads in the group that will meet at the barrier.

Parameters

Barrier: This is the barrier that will be initialized.

Attr: The Attr parameter is the barrier attribute that is used to define the initial configuration of the barrier when it is created. Under Linux, the only use for a barrier attribute is to determine if that barrier may be used between processes.

Count: The Count parameter is the number of threads that will meet at this barrier. Setting this parameter to 0 will result in the EINVAL error code returning from the function.

Return Value

If the function is successful, it will return 0. This function will only fail if the Count parameter was set to zero, in which case the function will return the EINVAL error code.

See Also

`pthread_barrier_destroy`, `pthread_barrier_wait`, `pthread_barrierattr_init`

Example

See Listing 6.21 - Using barriers.

pthread_barrier_wait LibC.pas*Syntax*

```
function pthread_barrier_wait(
  var Barrier: TPthreadBarrier
):Integer;
```

Description

The pthread_barrier_wait function will cause the current thread to wait on a barrier until all threads that need to have reached the barrier point. Once the function returns, all threads have successfully reached the barrier.

Parameters

Barrier: This is a barrier variable that threads will wait on.

Return Value

If the calling thread is not the last thread to reach the barrier, the function will return 0 and will wait for all the threads to reach the barrier by calling pthread_barrier_wait on the Barrier variable. If the calling thread was the last one to call pthread_barrier_wait, the function will return the PTHREAD_BARRIER_SERIAL_THREAD constant.

See Also

pthread_barrier_init, pthread_barrier_destroy

Example

See Listing 6.21 - Using barriers.

pthread_barrierattr_destroy LibC.pas*Syntax*

```
function pthread_barrierattr_destroy(
  var Barrier: TPthreadBarrier
):Integer;
```

Description

The pthread_barrierattr_destroy function releases any allocated resources by a call to the pthread_barrierattr_init function. Under Linux, this function does nothing and always returns successful.

Parameters

Barrier: This is the barrier attribute to destroy.

Return Value

This function will always return 0.

See Also

`pthread_barrierattr_init`, `pthread_barrierattr_getpshared`, `pthread_barrierattr_setpshared`

pthread_barrierattr_getpshared **LibC.pas**

Syntax

```
function pthread_barrierattr_getpshared(
  const Attr: TPthreadBarrierAttribute;
  var ProcessShared: Integer
):Integer;
```

Description

The `pthread_barrierattr_getpshared` function returns whether or not a barrier attribute has set the process sharing flag. This value will determine if barriers created with the barrier attribute can be used between processes.

Parameters

Attr: This is the barrier attribute to analyze.

ProcessShared: The ProcessShared parameter is an integer variable, which after the function call will hold a value that represents if the barrier can be shared between processes or not. If the barrier can be shared between processes, this parameter will be equal to the constant `PTHREAD_PROCESS_SHARED`. If the barrier cannot be shared between processes, this parameter will be equal to the constant `PTHREAD_PROCESS_PRIVATE`.

Return Value

This function always returns 0.

See Also

`pthread_barrierattr_destroy`, `pthread_barrierattr_init`, `pthread_barrierattr_setpshared`

pthread_barrierattr_init **LibC.pas**

Syntax

```
function pthread_barrierattr_init(
  var Barrier: TPthreadBarrier
):Integer;
```

Description

According to the POSIX standard, the `pthread_barrierattr_init` function initializes a barrier attribute record with the default values for a barrier on the system. On Linux, however, this function will simply make the attribute be used only within the current process.

Parameters

Barrier: This parameter is the barrier attribute to initialize.

Return Value

This function will always return 0.

See Also

pthread_barrierattr_destroy, pthread_barrierattr_getpshared, pthread_barrierattr_setpshared

pthread_barrierattr_setpshared LibC.pas**Syntax**

```
function pthread_barrierattr_setpshared(
  var Attr: TBarrierAttribute;
  ProcessShared: Integer
):Integer;
```

Description

The pthread_barrierattr_setpshared function changes the process sharing facility for a barrier attribute so that barriers created with this attribute can or cannot be shared between processes.

Parameters

Attr: This is the barrier attribute whose process sharing field will be changed.

ProcessShared: The ProcessShared parameter determines if the barrier can be shared between processes. If the process can be shared between processes, this value will be PTHREAD_PROCESS_SHARED. If the value cannot be shared between processes, the ProcessShared value will be PTHREAD_PROCESS_PRIVATE.

Return Value

If the function is successful, it will return 0. The only reason the function would be unsuccessful is if the value passed to ProcessShared was not PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE, and in that case the function would return the EINVAL error code.

See Also

pthread_barrierattr_destroy, pthread_barrierattr_getpshared, pthread_barrierattr_init

pthread_cancel LibC.pas**Syntax**

```
function pthread_cancel(
  ThreadID: TThreadID
):Integer;
```

Description

The use of the `pthread_cancel` function is to terminate the thread represented by `ThreadID` as soon as possible. In most cases, the thread will be terminated instantly unless the thread has used `pthread_setcancelstate` on the thread to disable the cancellation of the thread, or the `pthread_setcanceltype` function has been used to delay the cancellation until a cancellation point has been reached. A cancellation point is when the thread that is to be canceled calls the `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `semwait`, or `sigwait` functions.

Parameters

`ThreadID`: This parameter is the handle of the thread that should be canceled.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code. The function will normally only fail if the value passed to the `ThreadID` does not represent a POSIX thread.

See Also

`pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel`

Example

Listing 6.27 - Using the `pthread_cancel` function

```
program pthread_cancel_Demo;

{$APPTYPE CONSOLE}
uses
  Libc,
  SysUtils;

var
  ThreadID: Cardinal;

function DisplayTheTimeEverySecond(Data: Pointer): integer; cdecl;
begin
  //This function will be executed in a separate thread
  repeat
    writeln('The Current Time is ', TimeToStr(Now));
    sleep(1000);
  until false;
end;

begin
  //Create the thread
  writeln('Creating the thread.');
```

`pthread_create(ThreadID, nil, DisplayTheTimeEverySecond, nil);`

```

  //Let's wait 5 seconds
  writeln('Thread 1 waiting 5 seconds.');
```

`sleep(5000);`

```
//Let's cancel the thread
writeln('Cancelling the thread.');
```

```
pthread_cancel(ThreadID);

//Let's wait another 5 seconds
writeln('Thread 1 waiting 5 seconds.');
```

```
sleep(5000);
end.
```

pthread_cond_broadcast LibC.pas

Syntax

```
function pthread_cond_broadcast(
  var Cond: TCondVar
):Integer;
```

Description

The purpose of a condition variable is to wait for a notification of a particular event. The condition variable is signaled to let you know that the specific event has occurred.

The `pthread_cond_broadcast` function signals the condition variable passed to as the `Cond` parameter, and all threads, which are waiting on that condition variable to be signaled, will resume. This function is different from the `pthread_cond_signal` function, which will resume only one thread.

Parameters

`Cond`: This is the condition variable that will be signaled. Once the condition variable is signaled, all threads that are waiting on this condition variable will be resumed.

Return Value

This function will always return 0.

See Also

`pthread_cond_signal`, `pthread_cond_wait`, `pthread_cond_timedwait`

Example

See Listing 6.16 - Broadcasting to a condition variable.

pthread_cond_destroy LibC.pas

Syntax

```
function pthread_cond_destroy(
  var Cond: TCondVar
):Integer;
```

Description

A condition variable is a synchronization mechanism that allows a thread to suspend itself and wait for something to signal the thread to awaken. A condition variable is normally used when the thread cannot proceed until some event occurs.

The `pthread_cond_destroy` function will destroy any resources allocated when a condition variable was initialized with the `pthread_cond_init` function. Under LinuxThreads no allocation of resources was done, and as a result, this function does nothing but ensure that no threads are still waiting for a condition variable to be signaled.

Parameters

Cond: This is the condition variable that needs all of its resources deallocated.

Return Value

If the function is successful, it will return 0. If the condition variable has threads waiting on it, the function will return the EBUSY error code.

***pthread_cond_init* LibC.pas**

Syntax

```
function pthread_cond_init(
  var Cond: TCondVar;
  CondAttr: PPthreadCondAttr
):Integer;
```

Description

A condition variable is a signaling synchronization mechanism that allows a thread to suspend itself and wait for something to signal it to awaken. The condition variable is normally used when the thread cannot proceed until some event occurs and then the thread can continue.

The `pthread_cond_init` function sets up a condition variable for use. The condition variable will be created with the attributes set up by the `CondAttr` parameter.

Parameters

Cond: This is the condition that will be initialized.

CondAttr: This parameter is a condition variable attribute that defines how the condition variable should be created. This parameter can also be nil to use the default attributes. On LinuxThreads, however, condition variable attributes are not needed, so it is often best to use nil for this parameter.

Return Value

This function will always return 0.

See Also

`pthread_cond_destroy`, `pthread_cond_broadcast`, `pthread_cond_signal`, `pthread_cond_timedwait`, `pthread_cond_wait`

Example

See Listing 6.10 - A Web server using thread pooling.

pthread_cond_signal LibC.pas***Syntax***

```
function pthread_cond_signal(
  var Cond: TCondVar
):Integer;
```

Description

When a condition variable is used to suspend a thread with the `pthread_cond_wait` or `pthread_cond_timedwait` function, the thread needs to be notified by another thread so that the suspended thread can awaken.

The purpose of the `pthread_cond_signal` is to notify a thread waiting on a condition variable to awaken. Although many threads may be waiting for a condition variable, using the `pthread_cond_signal` function will only wake up a single thread, although the thread that is woken cannot be determined effectively.

To awaken more than one thread waiting for a condition variable to be signaled, use the `pthread_cond_broadcast` function.

Parameters

`Cond`: This is the condition variable that will be signaled. Once the condition variable is signaled, a thread that has been suspended as a result of a call to the `pthread_cond_signal` will wake from its suspended state and resume.

Return Value

This function will always return 0.

See Also

`pthread_cond_broadcast`, `pthread_cond_wait`, `pthread_cond_timedwait`

Example

See Listing 6.10 - A Web server using thread pooling.

pthread_cond_timedwait LibC.pas***Syntax***

```
function pthread_cond_timedwait(
  var Cond: TCondVar;
  var Mutex: TRTLCriticalSection;
  const AbsTime: TTimeSpec
):Integer;
```

Description

The `pthread_cond_timedwait` function is similar to the `pthread_cond_wait` function in that it will cause the calling thread to suspend until the condition variable is signaled and a lock can be obtained on the mutex passed to the function. The main difference between this function and the `pthread_cond_wait` function is that this function will only suspend up to a

time specified in the `AbsTime` parameter. If the condition variable cannot be obtained in that time, the function will return with the `ETIMEDOUT` error code.

Parameters

`Cond`: This is a valid condition variable that will be waited on.

`Mutex`: This parameter is an unlocked mutex that is used to ensure that when a signal has been sent, only one thread will awaken at a given time. In this way, different threads can have different condition variables but share the same mutex.

`AbsTime`: The `AbsTime` parameter defines how long the thread will attempt to awake in seconds and milliseconds. If this timeout period is reached, the function will return the `ETIMEDOUT` error code.

Return Value

If the function is successful, and the thread waits for the condition variable, the function will return 0. If the timeout period is reached, the function will return the `ETIMEDOUT` error code.

`pthread_cond_wait` LibC.pas

Syntax

```
function pthread_cond_wait(
  var Cond: TCondVar;
  var Mutex: TRTLCriticalSection
):Integer;
```

Description

A condition variable is a signaling synchronization mechanism that allows a thread to suspend itself and wait for some other thread to signal the condition variable and wake the calling thread. The condition variable is normally used when the thread cannot proceed until some event occurs.

The `pthread_cond_wait` function will unlock the mutex passed as the `Mutex` parameter and cause the current thread to block until another thread signals the condition variable by a call to either `pthread_cond_signal` or `pthread_cond_broadcast`. This will tell the thread to only proceed past this function call with a lock on the mutex. When another thread signals the condition variable `Cond`, using either `pthread_cond_signal` or `pthread_cond_broadcast`, and the mutex passed to the `Mutex` parameter can be locked, this function will return.

Parameters

`Cond`: This is a valid condition variable that will be waited on.

`Mutex`: This parameter is an unlocked mutex that is used to ensure that when a signal has been sent only one thread will awaken at a time. In this way, different threads can have different condition variables but share the same mutex.

Return Value

This function will always return 0.

See Also

pthread_cond_broadcast, pthread_cond_signal, pthread_cond_timedwait

Example

See Listing 6.10 - A Web server using thread pooling.

pthread_condattr_destroy LibC.pas*Syntax*

```
function pthread_condattr_destroy(
  var Attr: TPthreadCondAttr
):Integer;
```

Description

The pthread_condattr_destroy function will destroy a condition variable attribute so that any resources allocated by the call to pthread_condattr_init can be cleaned up. Under LinuxThreads, however, the condition variable attribute is simply a record, and as a result, this function does not actually do anything.

Parameters

Attr: This is the TPthreadCondAttr condition variable attribute that will be deinitialized.

Return Value

This function will always return 0.

See Also

pthread_condattr_init, pthread_condattr_getpshared, pthread_condattr_setpshared

pthread_condattr_getpshared LibC.pas*Syntax*

```
function pthread_condattr_getpshared(
  var Attr: TPthreadCondAttr;
  var ProcessShared: Integer
):Integer;
```

Description

The pthread_condattr_getpshared function will return whether or not the condition variable attribute indicates that the process should be shared. In LinuxThreads, the sharing of condition variables is not allowed, and as a result, this function will always return that the condition variable cannot be shared between processes.

Parameters

Attr: This parameter is a TPthreadCondAttr condition variable attribute.

ProcessShared: After the function call, the integer variable passed as the `ProcessShared` parameter will contain whether or not the condition variable can be shared between processes. The value `PTHREAD_PROCESS_SHARED` should be returned if the condition variable can be shared between processes or `PTHREAD_PROCESS_PRIVATE` if the condition variable cannot be shared. As LinuxThreads does not allow the sharing of condition variables between processes, the value returned to this parameter will always be `PTHREAD_PROCESS_PRIVATE`.

Return Value

This function will always return 0.

See Also

`pthread_condattr_init`, `pthread_condattr_destroy`, `pthread_condattr_setpshared`

pthread_condattr_init LibC.pas

Syntax

```
function pthread_condattr_init(
  var Attr: TPthreadCondAttr
):Integer;
```

Description

The `pthread_condattr_init` function initializes a condition variable attribute to the default values, so that condition variables created with the condition variable attribute will have the default values. Under LinuxThreads, this function will simply state that the condition variables created cannot be shared between processes.

Parameters

Attr: The `Attr` parameter is a `TPthreadCondAttr` variable that will be initialized.

Return Value

This function will always return 0.

See Also

`pthread_condattr_destroy`, `pthread_condattr_getpshared`, `pthread_condattr_setpshared`

pthread_condattr_setpshared LibC.pas

Syntax

```
function pthread_condattr_setpshared(
  var Attr: TPthreadCondAttr;
  ProcessShared: Integer
):Integer;
```


Description

The `pthread_condattr_setpshared` function defines whether or not a condition variable created with the condition variable attribute `Attr` can be shared between different processes. LinuxThreads does not implement this functionality, however, so only condition variables that are used within the same process are allowed.

Parameters

`Attr`: This is the condition variable attribute that will have its process sharing information altered.

`ProcessShared`: The `ProcessShared` parameter determines if condition variables created with the condition variable attribute `Attr` can be shared between processes. Setting this value to `PTHREAD_PROCESS_SHARED` should allow sharing between processes and `PTHREAD_PROCESS_PRIVATE` allows the condition variable to only be used internally. Only the `PTHREAD_PROCESS_PRIVATE` value is accepted under Linux. All other values will result in the function returning an error.

Return Value

If the function is successful, it will return 0. The function will only fail if the value is anything but `PTHREAD_PROCESS_PRIVATE`. If the value is `PTHREAD_PROCESS_SHARED`, the function will return the `ENOSYS` error code to denote that the sharing of condition variables is not implemented on this system. Any other value will result in the `EINVAL` error code being returned.

See Also

`pthread_condattr_init`, `pthread_condattr_destroy`, `pthread_condattr_getpshared`

pthread_create LibC.pas*Syntax*

```
function pthread_create(
  var ThreadID: TThreadID;
  Attr: PThreadAttr;
  StartRoutine: TPThreadFunc;
  Arg: Pointer
):Integer;
```

Description

The `pthread_create` function creates a new POSIX thread. When the new thread is created, the `StartRoutine` function is called with the `Arg` parameter being used as the parameter sent to it. The new thread will also be created as defined by the thread attribute passed into the `Attr` parameter. After the function call, the thread's handle will be stored in the `ThreadID` parameter. The newly created thread can be referenced by the `ThreadID` parameter in other functions that access and manipulate the thread.

Parameters

ThreadID: This parameter is an integer variable that after a successful function call will hold the handle to the thread. This thread handle is used in many thread functions where one thread wants to perform an action or request some information about another thread.

Attr: This parameter is a pointer to a `TThreadAttr` thread attribute record. The thread attribute defines the qualities for the thread. For more information on thread attributes, see the section titled “Thread Attributes” earlier in this chapter. If `nil` is passed to this parameter, the default attributes are used for the thread.

StartRoutine: This is a function that takes a single pointer parameter, returns an integer, and uses the `cdecl` calling convention. When the new thread is created, the first function that is called is this `StartRoutine` function. This parameter also works in conjunction with the `Arg` parameter so that data can be passed to the newly created thread.

type

```
TPThreadFunc = function(Parameter: Pointer): Integer; cdecl;
```

Arg: When the `StartRoutine` function is called in the newly created thread, the only parameter passed to the `StartRoutine` function is the `Arg` parameter.

Return Value

If the function is successful, it will return 0. The function will return an error code when it is unsuccessful. An unsuccessful result will normally be due to insufficient memory resources or because the maximum number of threads on the system has been reached.

See Also

`pthread_attr_init`, `pthread_cancel`

Example

See Listing 6.1 - Creating a thread using the `pthread_create` function.

`pthread_detach` LibC.pas

Syntax

```
function pthread_detach(
  ThreadID: TThreadID
):Integer;
```

Description

The `pthread_detach` function disables the ability of a thread to be joined by the use of the `pthread_join` function. As a result, when the thread terminates, the resources of the thread will be immediately released. Threads can be detached by using the `pthread_attr_setdetachstate` function on the attribute that was used to create the thread to specify that the thread will not be joined. This function is mainly used when the thread was created with a `nil` thread attribute, which means that the default settings for the POSIX thread will be used. By default, threads are joinable when they are created, unless specified otherwise.

Parameters

ThreadID: This is the handle of the thread that is attempting to be detached.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code. This function will fail if the thread handle passed into the ThreadID function is invalid or the thread is already detached.

See Also

pthread_join, pthread_attr_getdetachstate, pthread_attr_setdetachstate

Example

Listing 6.28 - Using the pthread_detach function

```

type
  PFileToCopy = ^TFileToCopy;
  TFileToCopy = record
    SourceFile: array[0..100] of char;
    DestinationFile: array[0..100] of char;
  end;

function BackgroundCopyFile(Data: Pointer): integer; cdecl;
var
  PFiles: PFileToCopy;
  buffer: array[0..1023] of char;
  BytesRead: integer;
  SourceFileDescriptor: integer;
  DestFileDescriptor: integer;
begin
  //Let's detach from the main thread
  pthread_detach(pthread_self);

  //Copy the file
  PFiles := Data;

  SourceFileDescriptor := open(PFiles^.SourceFile, O_RDONLY);
  DestFileDescriptor := open(PFiles^.DestinationFile, O_WRONLY or O_CREAT);
  BytesRead := __read(SourceFileDescriptor, buffer, 1024);
  while BytesRead > 0 do
    begin
      __write(DestFileDescriptor, buffer, BytesRead);
      BytesRead := __read(SourceFileDescriptor, buffer, 1024);
    end;
    __close(DestFileDescriptor);
    __close(SourceFileDescriptor);

    Result := 0;
  end;

procedure CopyFileInBackground(SourceFile, DestFile: string);
var
  FileDetails: PFileToCopy;
  MyThread: ^pthread_t;

```

```

begin
    //This function will copy a file in a background thread
    new(FileDetails);
    StrPCopy(FileDetails^.SourceFile, SourceFile);
    StrPCopy(FileDetails^.DestinationFile, DestFile);

    //Create the IDs.
    new(MyThread);

    //Let's create the thread now.
    pthread_create(MyThread^, nil, BackgroundCopyFile, FileDetails);
end;

```

pthread_equal LibC.pas

Syntax

```

function pthread_equal(
    Thread1ID: TThreadID;
    Thread2ID: TThreadID
):Integer;

```

Description

The `pthread_equal` function examines two thread handles and determines if they actually point to the same thread. On Linux, this is the same as using:

```

if Thread1ID = Thread2ID then
begin
    //They are equal
end;

```

except using this function is more portable to different operating systems that implement POSIX threads.

Parameters

Thread1ID: This is a handle to a POSIX thread.

Thread2ID: This is a handle to another POSIX thread.

Return Value

If the two threads are equal, the function will return a non-zero value. If the handles passed into Thread1ID and Thread2ID do not represent the same thread, 0 will be returned.

See Also

`pthread_self`

Example

See Listing 6.25 - Using the `GetCurrentThreadID` function.

pthread_exit LibC.pas***Syntax***

```

procedure pthread_exit(
  RetVal: Pointer
);

```

Description

The `pthread_exit` function terminates the current thread, executes all cleanup code for the thread, and sets the thread's return code to whatever value was passed to the `RetVal` parameter. The value passed into `RetVal` is what is retrieved when another thread is waiting for the thread to terminate using the `pthread_join` function.

Parameters

`RetVal`: This is the value that will be used as a return value to a thread that is waiting for the current thread to terminate using the `pthread_join` function.

See Also

`pthread_join`

Example**Listing 6.29 - Using the `pthread_exit` function**

```

function CalculateFactorialInThread(Data: Pointer): integer; cdecl;
var
  N, counter: integer;
  ReturnValue: integer;
begin
  //Here we will calculate n!. N is passed in as a typecast Pointer
  N := integer(Data);

  ReturnValue := 1;

  //Calculate n!
  for counter := 2 to N do
    ReturnValue := ReturnValue * counter;

  pthread_exit(Pointer(ReturnValue));
end;

procedure TfrmFactorial.btnCalculateClick(Sender: TObject);
var
  N, FactorialResult: integer;
  ThreadID: Cardinal;
  ResultingPointer: Pointer;
begin
  //Calculate the factorial in the background.
  N := seNValue.Value;
  pthread_create(ThreadID, nil, CalculateFactorialInThread, Pointer(N));

  //Let's wait for the thread

```

```

    ResultingPointer := @FactorialResult;
    pthread_join(ThreadID, ResultingPointer);
    Showmessage('The result of the calculation is '+
        inttostr(FactorialResult));
end;

```

pthread_getschedparam LibC.pas

Syntax

```

function pthread_getschedparam(
    TargetThreadID: TThreadID;
    var Policy: Integer;
    var Param: TSchedParam
):Integer;

```

Description

The `pthread_getschedparam` function retrieves the thread policy and priority of the thread specified by the `TargetThreadID` parameter.

Parameters

TargetThreadID: This parameter is a handle to a valid POSIX thread.

Policy: The `Policy` variable is an integer variable that will store the scheduling policy used for the thread after the function call has executed. The value returned will be `SCHED_RR` if the thread is using a real-time, round robin scheduling algorithm, `SCHED_FIFO` if the thread is using a real-time first-in/first-out algorithm, or `SCHED_OTHER` if scheduling is explicitly set.

```

const
    SCHED_OTHER    = 0;
    SCHED_FIFO     = 1;
    SCHED_RR       = 2;

```

Param: This parameter is a `TSchedParam` variable that after a successful call will contain the thread priority. The thread priority is dependent on the policy used. Minimum and maximum values for the priority can be obtained by using the `sched_get_priority_min` and `sched_get_priority_max` functions.

Return Value

If the function is successful, it will return 0 and place the thread's policy in the `Policy` parameter and the thread's priority in the `Param` parameter. If the function is unsuccessful, it will return a valid error code, which will normally be the result of an invalid value being passed to the `TargetThreadID` parameter.

See Also

`pthread_attr_getschedpolicy`, `pthread_attr_getschedparam`, `pthread_attr_setschedpolicy`, `pthread_attr_setschedparam`

*Example***Listing 6.30 - Using the pthread_getschedparam function**

```

procedure TfrmParamInfo.FormCreate(Sender: TObject);
var
    Policy: integer;
    Priority: TSchedParam;
begin
    //Display the information about the current (main) thread
    pthread_getschedparam(pthread_self, Policy, Priority);

    case Policy of
        SCHED_RR: mmInfo.Lines.Add('Policy: Round Robin. ');
        SCHED_FIFO: mmInfo.Lines.Add('Policy: First-in/First-out (FIFO) ');
        SCHED_OTHER: mmInfo.Lines.Add('Policy: Other ');
    end;

    mmInfo.Lines.Add('Priority: '+inttostr(Priority.sched_priority));
end;

```

pthread_getspecific LibC.pas*Syntax*

```

function pthread_getspecific(
    Key: TPTThreadKey
):Pointer;

```

Description

In POSIX thread applications, threads can contain data specific to the calling thread. This thread-specific data is accessed using a TPTThreadKey variable that is used to obtain the data that is specific for a thread. Unlike a variable, though, thread-specific data needs to be accessed by the interface functions pthread_getspecific and pthread_setspecific.

The pthread_getspecific function is used to obtain thread-specific data from a valid thread key that was previously set by a call to the pthread_setspecific function.

Parameters

Key: This is the TPTThreadKey variable that is used as a key value to access the thread-specific data.

Return Value

If the function is successful, it will return a valid pointer to the thread-specific data. If the thread-specific data has not been set by a previous call to pthread_setspecific, the Key parameter is not a valid key and the function will return nil.

See Also

pthread_setspecific, pthread_key_create, pthread_key_delete

Example

See Listing 6.23 - A multi-threaded version library using thread keys.

pthread_join LibC.pas***Syntax***

```
function pthread_join(
  ThreadID: TThreadID;
  ThreadReturn: PPointer
):Integer;
```

Description

The `pthread_join` function will block until the thread represented by the thread handle passed into the `ThreadID` parameter terminates. This function can also be used to obtain information on how the POSIX thread terminated. This function is similar to the wait functions when used on a process, except that `pthread_join` only works on threads. You should note that the thread the current thread is waiting on should be joinable by using the `pthread_attr_setdetachstate` function. It is also worth noting that only one thread can wait on a thread at one time.

Parameters

`ThreadID`: This is the handle of the POSIX thread that the function should wait for.

`ThreadReturn`: This parameter is a pointer to another pointer. This second pointer points to the return code of the `TPThreadFunc` function that was the starting function for the thread in the call to `pthread_create`. If you are not interested in the exit code of the thread function, this function can be `nil`, in which case the exit code is not returned.

Return Value

If the function is successful, it will return 0 and the `ThreadReturn` parameter will contain the exit code of the function if the `ThreadReturn` parameter is not `nil`. If the function is unsuccessful, the function will return a valid error code.

See Also

`pthread_create`, `pthread_detach`, `pthread_cancel`

Example

See Listing 6.5 - Waiting for a POSIX thread to finish.

pthread_key_create LibC.pas***Syntax***

```
function pthread_key_create(
  var Key: TPThreadKey;
  DestrFunction: TKeyValuePairDestructor
):Integer;
```


Description

The `pthread_key_create` function will create a thread-specific key. This thread-specific key allows multiple threads to reference a single variable to obtain data that is specific for that thread. All threads will have access to this newly created thread key, but each thread will store and read its own data by referencing the `TPThreadKey` variable in calls to `pthread_getspecific` and `pthread_setspecific` to access thread-specific data. As well as creating the key, this function also specifies a destructor function `DestrFunction` that is used to remove any thread-specific data when the thread exits or is terminated.

Parameters

Key: The `Key` parameter is a `TPThreadKey` variable that is used to access information specific to a thread.

DestrFunction: The `DestrFunction` parameter is a function that will be used to release any memory that is stored at the thread's specific data location. Because thread-specific data only contains a pointer value, often the thread-specific data will contain a pointer to a block used for the specific thread. Using this parameter can ensure that the memory is successfully removed. Specifying `nil` for this parameter will mean that no destructor function will be called.

Return Value

If the function is successful, it will return 0. The function will usually only be unsuccessful if the number of keys created exceeds the maximum allowed. In this case, the function will return `EAGAIN`.

See Also

`pthread_key_delete`, `pthread_getspecific`, `pthread_setspecific`

Example

See Listing 6.23 - A multi-threaded version library using thread keys.

pthread_key_delete LibC.pas*Syntax*

```
function pthread_key_delete(
  Key: TPThreadKey
):Integer;
```

Description

The `pthread_key_delete` function will remove a thread-specific data key that was previously allocated by a call to `pthread_key_create`. Calling this function will not cause the destructor function passed to `pthread_key_create` to be called on the active threads. If the thread-specific data key is still in use, this function will return `invalid`.

Parameters

Key: This is a TPTThreadKey variable that represents the thread's specific data key to remove.

Return Value

If the function is successful, it will return 0. The function will fail if the key is currently in use or if the key was never created, in which case the function will return the EINVAL error code.

See Also

pthread_key_create, pthread_getspecific, pthread_setspecific

Example

See Listing 6.23 - A multi-threaded version library using thread keys.

pthread_kill LibC.pas**Syntax**

```
function pthread_kill(
  ThreadID: TThreadID;
  SigNum: Integer
):Integer;
```

Description

The pthread_kill function is used to send a signal, represented by SigNum, to another thread identified by the ThreadID parameter. This is similar to the kill function but is used to communicate between threads instead of processes. For more information on working with threads and signals, see the section titled "Signals and Threads."

Parameters

ThreadID: This parameter is the thread to which the signal should be sent.

SigNum: This parameter represents the signal that will be sent to the thread.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

kill, pthread_sigmask

Example

See Listing 6.24 - Extending the behavior of TThread.

pthread_kill_other_threads_np LibC.pas***Syntax***

```
procedure pthread_kill_other_threads_np
```

Description

This function will kill all the threads in the current process except the calling thread. This function is mainly used when performing one of the exec functions to execute a process. This function is specific to the Linux implementation of POSIX threads. Killing threads using this function will bypass any cleanup handler code set up using the `_pthread_cleanup_push` function or other cleanup mechanism.

See Also

```
pthread_create, exec, pthread_cancel, pthread_setcancelstate, _pthread_cleanup_push
```

pthread_mutex_destroy LibC.pas***Syntax***

```
function pthread_mutex_destroy(  
  var Mutex: TRTLCriticalSection  
):Integer;
```

Description

The `pthread_mutex_destroy` function releases any memory that was used when the mutex was originally created. On the Linux implementation of POSIX threads, the function is used to determine that the mutex is not locked and no threads are waiting on the thread.

Parameters

Mutex: This is the mutex that will be destroyed.

Return Value

If the function is successful, it will return 0. The function will return an error code when the mutex is still in use by one or more threads.

See Also

```
pthread_mutex_init, pthread_mutexattr_init
```

Example

See Listing 6.8 - Protecting the resource with a mutex.

pthread_mutex_init LibC.pas***Syntax***

```
function pthread_mutex_init(  
  var Mutex: TRTLCriticalSection;  
  var Attr: TMutexAttribute
```

```
);Integer;
```

Description

The `pthread_mutex_init` function is used to initialize a mutex variable. Additional attributes for the newly created mutex can be used by passing a mutex attribute variable to the `Attr` parameter. After the function call the mutex can be successfully used by multiple threads.

Parameters

Mutex: The `Mutex` parameter is a `TRTLCriticalSection` variable that contains the mutex that will be initialized.

Attr: This parameter is a `TMutexAttribute` variable that defines the attributes that will be used when the thread is created.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_mutex_destroy`, `pthread_mutex_lock`, `pthread_mutex_timedlock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`

Example

See Listing 6.8 - Protecting the resource with a mutex.

***pthread_mutex_lock* LibC.pas**

Syntax

```
function pthread_mutex_lock(
  var Mutex: TRTLCriticalSection
):Integer;
```

Description

The `pthread_mutex_lock` function requests that the current thread obtain a lock on a mutex. As only one thread can have access to a mutex at a given time, if another thread has locked the mutex the function will block until the thread is able to obtain a lock on the mutex. When the function returns, the current thread will have the lock on the mutex.

Once the thread has a lock on the mutex, subsequent calls can be made to the `pthread_mutex_lock` function on the mutex. Depending on the type, the mutex will react in different ways. If the mutex is a fast mutex, the thread will wait until the mutex becomes available. As it is already taken by the current thread, this will cause a deadlock and is not recommended. If the mutex is a recursive mutex, it will increase the counter for the mutex and the same number of calls to the `pthread_mutex_unlock` is required to unlock the mutex. If the mutex is an error-checking mutex, the new calls to `pthread_mutex_lock` will result in the function returning the `EDEADLK` error code. For more information on mutex types, see the `pthread_mutexattr_settype` function.

Parameters

Mutex: This is the mutex that is to be locked on the current thread.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

pthread_mutex_unlock, pthread_mutex_trylock, pthread_mutex_timedlock

Example

See Listing 6.8 - Protecting the resource with a mutex.

pthread_mutex_timedlock LibC.pas**Syntax**

```
function pthread_mutex_timedlock(
  var Mutex: TRTLCriticalSection;
  const AbsTime: timespec
):Integer;
```

Description

The pthread_mutex_timedlock function is almost identical in use to the pthread_mutex_lock function, with the only difference being that pthread_mutex_timedlock will wait for the time interval specified in the AbsTime parameter to pass. This function only works for mutexes where the type attribute is specified as PTHREAD_MUTEX_TIMED_NP.

Parameters

Mutex: This is the mutex that is to be locked on the current thread.

AbsTime: This is the time in seconds and nanoseconds that the thread can wait before returning.

```
type
  timespec = record
    tv_sec: Longint; //Seconds
    tv_nsec: Longint; //Nanoseconds
  end;
```

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock, pthread_mutexattr_settype



Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code. The function is likely to fail if the mutex record is invalid or the thread that is calling this function does not own the lock on the mutex.

See Also

`pthread_mutex_lock`, `pthread_mutex_timedlock`, `pthread_mutex_trylock`

Example

See Listing 6.8 - Protecting the resource with a mutex.

`pthread_mutexattr_destroy` LibC.pas**Syntax**

```
function pthread_mutexattr_destroy(
  var Attr: TMutexAttribute
):Integer;
```

Description

The `pthread_mutexattr_destroy` function should be used to remove any resources used in the creation of a mutex attribute being created by the `pthread_mutexattr_init` function. As a mutex attribute is only a record, this function on Linux does nothing but return successfully every time.

Parameters

Attr: This is the mutex attribute to destroy.

Return Value

On the Linux implementation of threads, this function will always return 0.

See Also

`pthread_mutexattr_init`, `pthread_mutexattr_getpshared`, `pthread_mutexattr_setpshared`, `pthread_mutexattr_gettype`, `pthread_mutexattr_settype`

Example

See Listing 6.8 - Protecting the resource with a mutex.

`pthread_mutexattr_getpshared` LibC.pas**Syntax**

```
function pthread_mutexattr_getpshared(
  var Attr: TMutexAttribute;
  var ProcessShared: Integer
):Integer;
```

Description

The `pthread_mutexattr_getpshared` function retrieves whether or not the mutex attribute can be shared between processes. The Linux implementation of POSIX threads, however, only allows mutexes to communicate within the same process.

Parameters

Attr: The `Attr` parameter is a valid mutex attribute.

ProcessShared: This is an integer variable parameter that stores whether or not the mutex is shared between processes. On normal POSIX systems, the available options for this parameter are `PTHREAD_PROCESS_SHARED` to state that the mutex is shared between processes or `PTHREAD_PROCESS_PRIVATE` to specify that the mutex can be used within the same process only. On Linux, this parameter will always return `PTHREAD_PROCESS_PRIVATE`.

Return Value

On Linux's implementation of POSIX threads, this function will always return `PTHREAD_PROCESS_PRIVATE`.

See Also

`pthread_mutexattr_setpshared`, `pthread_mutexattr_init`, `pthread_mutexattr_destroy`

`pthread_mutexattr_gettype` LibC.pas

Syntax

```
function pthread_mutexattr_gettype(
  var Attr: TMutexAttribute;
  var Kind: Integer
):Integer;
```

Description

The `pthread_mutexattr_gettype` function returns the mutex type field from a mutex attribute record. There are three kinds of mutexes that can be used with threads on Linux. These are fast mutexes, recursive mutexes, and error-checking mutexes.

Fast mutexes are the most efficient to work with, but they come with the downside that the mutex can be unlocked by a thread that does not own the mutex. Recursive mutexes allow a mutex to be locked multiple times by a single thread with a call to one of the `pthread_mutex_lock` functions. In order for the thread to be unlocked, the thread must call the `pthread_mutex_unlock` function an equal amount of times. Error-checking mutexes are used for debugging/testing so that when a thread tries to obtain a lock on the mutex, the mutex will return an error and cause no deadlock.

Parameters

Attr: This is the mutex attribute record to work with.

Kind: This parameter is an integer variable that the mutex type will be stored in after the function call. If the type of mutex in the `Attr` parameter is a fast mutex, this parameter will

contain `PTHREAD_MUTEX_FAST_NP`. If the mutex type in the `Attr` parameter is a recursive mutex, this parameter will contain `PTHREAD_MUTEX_RECURSIVE_NP`. If the mutex is an error-checking mutex, this parameter will contain `PTHREAD_MUTEX_ERRORCHECK_NP`.

Return Value

This function will always return 0.

See Also

`pthread_mutexattr_settype`, `pthread_mutexattr_init`

pthread_mutexattr_init LibC.pas

Syntax

```
function pthread_mutexattr_init(
  var Attr: TMutexAttribute
):Integer;
```

Description

The `pthread_mutexattr_init` function initializes the mutex attribute with the default information. On Linux, this will set the mutex attribute to be a fast mutex, which is the quickest form of thread mutex, and the mutex can only be shared within the process.

Parameters

`Attr`: This parameter is a mutex variable that is to be set to the default values.

Return Value

This function will always return 0.

See Also

`pthread_mutexattr_destroy`, `pthread_mutexattr_getpshared`, `pthread_mutexattr_gettype`, `pthread_mutexattr_setpshared`, `pthread_mutexattr_settype`, `pthread_mutex_init`

Example

See Listing 6.8 - Protecting the resource with a mutex.

pthread_mutexattr_setpshared LibC.pas

Syntax

```
function pthread_mutexattr_setpshared(
  var Attr: TMutexAttribute;
  ProcessShared: Integer
):Integer;
```

Description

The `pthread_mutexattr_setshared` function defines whether or not a mutex can be shared between processes. On Linux, mutexes cannot be shared between processes so this function is effectively useless.

Parameters

Attr: This parameter is the mutex that wishes to have its attributes changed.

ProcessShared: This value specifies whether or not the mutex can be shared between processes. The available options are `PTHREAD_PROCESS_SHARED` to denote that mutexes can be shared between processes or `PTHREAD_PROCESS_PRIVATE` to specify that the mutex can only be used within the current process. As Linux does not support the sharing of mutexes between processes, using any value other than `PTHREAD_PROCESS_PRIVATE` will result in an error code being returned from the function.

Return Value

The function is only successful when `PTHREAD_PROCESS_PRIVATE` is used in the `ProcessShared` parameter. All other parameter options will result in an error occurring.

See Also

`pthread_mutexattr_getshared`

`pthread_mutexattr_settype` LibC.pas

Syntax

```
function pthread_mutexattr_settype(
  var Attr: TMutexAttribute;
  Kind: Integer
):Integer;
```

Description

The `pthread_mutexattr_settype` function sets the type of the mutex. There are three kinds of mutexes that can be used with POSIX threads on Linux. These are fast mutexes, recursive mutexes, and error-checking mutexes.

Fast mutexes are the most efficient to work with, but they come with the downside that the mutex can be unlocked by a thread that does not own the mutex. Recursive mutexes allow a mutex to be locked multiple times by a single thread with a call to one of the `pthread_mutex_lock` functions, and in order for the thread to be unlocked, the thread must call the `pthread_mutex_unlock` function an equal amount of times. Error-checking mutexes are used for debugging/testing so that when a thread tries to obtain a lock on the mutex, the mutex will return an error and cause no deadlock.

Parameters

Attr: This is the mutex attribute that will have its type set.

Kind: This parameter specifies the type of mutex that will be created. The possible values for this function are `PTHREAD_MUTEX_FAST_NP` for a fast mutex, `PTHREAD_`

MUTEX_RECURSIVE_NP for a recursive mutex, or PTHREAD_MUTEX_ERRORCHECK_NP for an error-checking mutex.

Return Value

If the function is successful, it will return 0. The function will only fail if the mutex type is not PTHREAD_MUTEX_FAST_NP, PTHREAD_MUTEX_RECURSIVE_NP, or PTHREAD_MUTEX_ERRORCHECK_NP.

See Also

pthread_mutexattr_init, pthread_mutexattr_gettype

pthread_once LibC.pas

Syntax

```
function pthread_once(
  var InitOnceSemaphore: Integer;
  InitRoutine: TInitOnceProc
):Integer;
```

Description

The pthread_once function executes a function specified in the InitRoutine function only once. The function that is called is only to be used to initialize some values that may be used in the thread. The first thread that calls the function will execute the procedure passed into the InitRoutine. Subsequent threads that call the pthread_once function will block until the InitRoutine procedure has been called.

Parameters

InitOnceSemaphore: This is an integer variable that is used to store whether or not the procedure passed into InitRoutine has been executed. This parameter should be set to the constant PTHREAD_ONCE_INIT before the pthread_once function is ever initialized.

InitRoutine: This is a procedure with no parameters using the cdecl calling convention. When using the pthread_once function, this procedure is guaranteed to be executed only once.

```
type
  TInitOnceProc = procedure; cdecl;
```

Return Value

This function will always return 0.

Example

See Listing 6.18 - Using the pthread_once function.

pthread_rwlock_destroy LibC.pas

Syntax

```
function pthread_rwlock_destroy(
```

```
var RWLock: TPthreadRWlock
):Integer;
```

Description

The `pthread_rwlock_destroy` function performs a cleanup operation on a read/write lock initialized with the `pthread_rwlock_init` function.

Parameters

RWLock: This is the read/write lock that should have its resources released.

Return Value

If the function is successful and the read/write lock can be released, it will return 0. If there are active locks on the read/write lock, the function will return `EBUSY`.

See Also

`pthread_rwlock_init`

Example

Listing 6.31 - Using the `pthread_rwlock_destroy` function

```
var
  rwLock: TPthreadRWlock;

function CleanupReadWriteLock: boolean;
begin
  Result := true;
  if pthread_rwlock_destroy(rwLock) = EBUSY then
  begin
    showmessage('There are still active locks on the read/write lock');
    Result := false;
  end;
end;
```

pthread_rwlock_init **LibC.pas**

Syntax

```
function pthread_rwlock_init(
  var RWLock: TPthreadRWlock;
  Attr: PPthreadRWlockAttribute
):Integer;
```

Description

The `pthread_rwlock_init` function initializes a read/write lock with the default attributes and allocates resources required to maintain the read/write lock. This function also sets the value of the read/write lock to a state where it appears that the read/write lock is not used by any threads. All read/write locks need to be passed to this function before the read/write lock can be used with the other read/write lock functions.

Parameters

RWLock: The RWLock parameter is a read/write lock variable that will be initialized.

Attr: The Attr parameter is a pointer to a read/write lock attribute variable that contains the settings that will be used when the read/write lock is initialized. When this parameter is nil, the default attributes for a read/write lock will be used.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

See Also

`pthread_rwlock_destroy`, `pthread_rwlockattr_init`

Example

See Listing 6.19 - Using a read/write lock.

`pthread_rwlock_rdlock` LibC.pas

Syntax

```
function pthread_rwlock_rdlock(  
  var RWLock: TPthreadRWlock  
):Integer;
```

Description

A read/write lock is a synchronization mechanism by which many threads can obtain read access for a resource, but only one thread can have write access to a resource.

The `pthread_rwlock_rdlock` function obtains a lock on the read/write lock for reading only. The locking mechanism for this function works by blocking only if a write lock is active on the read/write lock. When the write block has been lifted, the read lock will be obtained and the calling thread can then read the data. This function will not return until the read lock is obtained.

Parameters

RWLock: This is a read/write lock variable that is used as the means of synchronization of a resource.

Return Value

This function will always return 0.

See Also

`pthread_rwlock_timedrdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_tryrdlock`, `pthread_rwlock_trywrlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`

Example

See Listing 6.19 - Using a read/write lock.

pthread_rwlock_timedrdlock LibC.pas***Syntax***

```
function pthread_rwlock_timedrdlock(
  var RWLock: TPthreadRWlock;
  const AbsTime: timespec
):Integer;
```

Description

The `pthread_rwlock_timedrdlock` function is similar to the `pthread_rwlock_rdlock` function in that it attempts to obtain a read lock on a read/write lock. The main difference is that the function will only attempt to obtain the lock for a period of time; otherwise, notification that the timeout was reached will be returned by the function.

Parameters

RWLock: This is the read/write lock variable on which you want to obtain a read lock.

AbsTime: The `AbsTime` parameter is a `timespec` value that defines the maximum time period to wait until the read lock can be obtained.

```
type
  timespec = record
    tv_sec: Longint;      //Seconds
    tv_nsec: Longint;    //Nanoseconds
  end;
```

Return Value

If the function is successful, it will return 0 and there will be another read lock on the read/write lock. If the timeout period was reached, the function will return the ETIMEDOUT error code. If the time specified was not in the range of 0..1,000,000,000, the function will return the EINVAL error code.

See Also

`pthread_rwlock_rdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_tryrdlock`, `pthread_rwlock_trywrlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`

pthread_rwlock_timedwrlock LibC.pas***Syntax***

```
function pthread_rwlock_timedwrlock(
  var RWLock: TPthreadRWlock;
  const AbsTime: timespec
):Integer;
```

Description

The `pthread_rwlock_timedwrlock` function is similar to the `pthread_rwlock_wrlock` function in that it attempts to obtain a write lock on a read/write lock. The main difference is

that the function will only attempt to obtain the lock for a period of time; otherwise, notification that the timeout was reached will be returned from the function.

Parameters

RWLock: This is the read/write lock variable on which you want to obtain a write lock.

AbsTime: The AbsTime parameter is a timespec value that defines the maximum time period to wait until the write lock can be obtained. The AbsTime parameter must have a positive value and not be greater than 1,000,000,000; otherwise, the function will return the EINVAL error code.

```
type
    timespec = record
        tv_sec: Longint;      //Seconds
        tv_nsec: Longint;    //Nanoseconds
    end;
```

Return Value

If the function is successful, it will return 0 and there will be a write lock on the read/write lock. If the timeout period was reached, the function will return the ETIMEDOUT error code. If the time specified was not in the range of 0..1,000,000,000, the function will return the EINVAL error code.

See Also

pthread_rwlock_rdlock, pthread_rwlock_timedrdlock, pthread_rwlock_tryrdlock, pthread_rwlock_trywrlock, pthread_rwlock_wrlock, pthread_rwlock_unlock

pthread_rwlock_tryrdlock LibC.pas

Syntax

```
function pthread_rwlock_tryrdlock(
    var RWLock: TPtrthreadRWlock
):Integer;
```

Description

The pthread_rwlock_tryrdlock function is similar to the pthread_rwlock_rdlock in that it attempts to obtain a read lock on a read/write lock. This function is different in that the function will return an EBUSY error code if the read lock cannot be obtained right away.

Parameters

RWLock: This is a read/write lock variable that is used as the means of synchronization of a resource.

Return Value

If the function is successful and the read lock can be obtained, it will return 0. If the read lock cannot be obtained right away, the function will return the EBUSY error code.

See Also

`pthread_rwlock_timedrdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_rdlock`,
`pthread_rwlock_trywrlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`

Example

See Listing 6.19 - Using a read/write lock.

pthread_rwlock_trywrlock LibC.pas*Syntax*

```
function pthread_rwlock_trywrlock(
  var RWLock: TPthreadRWlock
):Integer;
```

Description

The `pthread_rwlock_trywrlock` function is similar to the `pthread_rwlock_wrlock` function in that it attempts to obtain a write lock on a read/write lock. This function is different in that it will return an `EBUSY` error code if the write lock cannot be obtained immediately.

Parameters

`RWLock`: This is a valid read/write lock variable to be used.

Return Value

If the function is successful and the write lock can be obtained, it will return 0. If the function could not obtain the lock immediately, it will return `EBUSY`.

See Also

`pthread_rwlock_timedrdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_rdlock`,
`pthread_rwlock_tryrdlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`

pthread_rwlock_unlock LibC.pas*Syntax*

```
function pthread_rwlock_unlock(
  var RWLock: TPthreadRWlock
):Integer;
```

Description

The `pthread_rwlock_unlock` function will release a lock on the read/write lock specified in the `RWLock` parameter. When the function is called, it will determine if the read/write lock is locked for reading or writing and will decrease the number of locks held on the reader of the write lock.

Parameters

`RWLock`: This is the read/write lock to unlock.

Return Value

If the function is successful, it will return 0. The function will return the EPERM error code if there are no locks on the read/write lock to release.

See Also

`pthread_rwlock_rdlock`, `pthread_rwlock_timedrdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_tryrdlock`, `pthread_rwlock_trywrlock`, `pthread_rwlock_unlock`

Example

See Listing 6.19 - Using a read/write lock.

`pthread_rwlock_wrlock` LibC.pas**Syntax**

```
function pthread_rwlock_wrlock(
  var RWLock: TPthreadRWlock
):Integer;
```

Description

The `pthread_rwlock_wrlock` function attempts to obtain write access to a resource protected by a read/write lock. A read/write lock works by allowing multiple threads read access but only allowing a single thread a write lock to the resource. When calling this function, if there are threads that have active read locks on the read/write lock, the function will not return until all the threads have relinquished their read locks, so that the write lock can be obtained.

Parameters

RWLock: This is the read/write lock that protects the resource that wishes to gain write access.

Return Value

This function will always return 0.

See Also

`pthread_rwlock_timedrdlock`, `pthread_rwlock_timedwrlock`, `pthread_rwlock_rdlock`, `pthread_rwlock_trywrlock`, `pthread_rwlock_unlock`

Example

See Listing 6.19 - Using a read/write lock.

`pthread_rwlockattr_destroy` LibC.pas**Syntax**

```
function pthread_rwlockattr_destroy(
  var Attr: TPthreadRWlockAttribute
):Integer;
```

Description

The `pthread_rwlockattr_destroy` function performs cleanup operations on any resources that were allocated with the `pthread_rwlockattr_init` function. The Linux implementation of this function does nothing and will always return successful.

Parameters

Attr: This is the read/write lock attribute on which the cleanup will be done.

Return Value

This function will always return 0.

See Also

`pthread_rwlockattr_init`, `pthread_rwlockattr_getpshared`, `pthread_rwlockattr_setpshared`

pthread_rwlockattr_getpshared LibC.pas

Syntax

```
function pthread_rwlockattr_getpshared(
  const Attr: TPthreadRWlockAttribute;
  var PShared: Integer
):Integer;
```

Description

The `pthread_rwlockattr_getpshared` function returns whether or not a read/write lock can be shared between multiple processes under Linux. As the Linux implementation of threads does not allow the sharing of read/write locks between processes, this function will always return that the read/write lock cannot be shared.

Parameters

Attr: This is a valid read/write lock attribute variable.

PShared: This parameter is an integer variable that determines if the process can be shared between processes. If the process can be shared, this parameter will contain `PTHREAD_PROCESS_SHARED`. If the process cannot be shared, this parameter will contain `PTHREAD_PROCESS_PRIVATE`. Under Linux, only private read/write locks are available.

Return Value

After calling this function, the PShared variable parameter will state whether the process can be shared or not and the function will return 0.

See Also

`pthread_rwlockattr_destroy`, `pthread_rwlockattr_init`

pthread_rwlockattr_init LibC.pas***Syntax***

```
function pthread_rwlockattr_init(
  var Attr: TPTHreadRWLockAttribute
):Integer;
```

Description

The `pthread_rwlockattr_init` function sets up a read/write lock attribute to the default values and sets up the lock so that it appears that no one is currently holding a lock. Under Linux's implementation of POSIX threads, this means that read/write locks created using this attribute will not be shared between processes and locks will favor write locks.

Parameters

Attr: This is the attribute that will have its values set to the default.

Return Value

This function will always return 0.

See Also

`pthread_rwlockattr_destroy`, `pthread_rwlockattr_getpshared`, `pthread_rwlockattr_setpshared`

Example**Listing 6.32 - Using read/write lock attributes**

```
procedure SetupReadWriteLock;
var
  RWLock: pthread_rwlock_t;
  RWLockAttribute: pthread_rwlockattr_t;
begin
  //Initialize the Attribute
  pthread_rwlockattr_init(RWLockAttribute);
  pthread_rwlockattr_setpshared(RWLockAttribute, PTHREAD_PROCESS_PRIVATE);
  pthread_rwlockattr_setkind_np(RWLockAttribute, PTHREAD_RWLOCK_DEFAULT_NP);

  //Now we will initialize the read/write lock
  pthread_rwlock_init(RWLock, @RWLockAttribute);
end;
```

pthread_rwlockattr_setpshared LibC.pas***Syntax***

```
function pthread_rwlockattr_setpshared(
  var Attr: TPTHreadRWLockAttribute;
  PShared: Integer
):Integer;
```

Description

The `pthread_rwlockattr_setpshared` function defines if a read/write lock can be shared between processes. Under the Linux implementation of POSIX threads, sharing read/write locks between processes is not allowed, so this function is essentially useless for Linux developers.

Parameters

Attr: This is the read/write lock attribute variable that will be used to initialize a read/write lock.

PShared: The `PShared` parameter is an integer that states whether the read/write lock can be shared between processes. Using `PTHREAD_PROCESS_PRIVATE` as the value means that the process cannot go outside process boundaries. Using `PTHREAD_PROCESS_SHARED` means that the process can be shared. As LinuxThreads only implements read/write locks within the current process, only the `PTHREAD_PROCESS_PRIVATE` value is accepted for this parameter. Any other value will result in an error.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, which would only be as a result of the `PShared` parameter not being the value `PTHREAD_PROCESS_PRIVATE`, it will return a valid error code.

See Also

`pthread_rwlockattr_destroy`, `pthread_rwlockattr_getpshared`, `pthread_rwlockattr_init`

***pthread_self* LibC.pas**

Syntax

```
function pthread_self: TThreadID;
```

Description

The `pthread_self` function returns the thread handle that represents the current thread.

Return Value

The function returns the thread handle of the current calling thread.

See Also

`GetCurrentThreadID`, `pthread_equal`

Example

See Listing 6.25 - Using the `GetCurrentThreadID` function.

***pthread_setcancelstate* LibC.pas**

Syntax

```
function pthread_setcancelstate(
```

```
State: Integer;
OldState: PInteger
):Integer;
```

Description

The `pthread_setcancelstate` function determines if the current thread will accept or ignore a cancellation request. By default, when threads are created, the threads will accept the cancellations. This function also obtains the previous cancellation state of the thread so that it can restore the state later.

Parameters

State: This is the cancellation state of the thread and will either be the constant `PTHREAD_CANCEL_ENABLE`, which means that the thread can be canceled, or `PTHREAD_CANCEL_DISABLE`, which means the thread will ignore any cancellation requests.

OldState: This parameter is a pointer to an integer that will return the previous cancellation state of the thread, which will be either `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`. If this parameter is `nil`, the previous cancellation state will not be returned.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, which would be due to a value other than `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`, the function will return a valid error code.

See Also

`pthread_setcanceltype`, `pthread_cancel`

Example

Listing 6.33 - Using the `pthread_setcancelstate` function

```
unit unitDisplayForm;

{
    This is an example of using the thread cancellation functions
    pthread_setcancelstate and pthread_setcanceltype.
    Use the controls to see the different effects it will have
    on the thread's cancellation.
}
interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs, Libc,
    QStdCtrls, QTypes, QExtCtrls;

type
    TfrmThreadDemo = class(TForm)
        cbCanCancel: TCheckBox;
        Label1: TLabel;
```

```

    cbCancelType: TComboBox;
    Label2: TLabel;
    btnCreateThread: TButton;
    btnCancelThread: TButton;
    tmrDisplayStatus: TTimer;
    lblStatusCounter: TLabel;
    procedure btnCreateThreadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure tmrDisplayStatusTimer(Sender: TObject);
    procedure btnCancelThreadClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    frmThreadDemo: TfrmThreadDemo;

    ThreadID: Cardinal;
    ValueCounter: integer = 0;

    CanCancelThread: boolean;
    ImmediateThreadCancel: boolean;

    ValueMutex: pthread_mutex_t;
    ValueMutexAttr: pthread_mutexattr_t;

implementation

{$R *.xfm}

function MyThreadFunction(Data: Pointer): integer; cdecl;
begin
    //Let's set up the thread
    if CanCancelThread then
    begin
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nil);
        //We will set the cancellation type also
        if ImmediateThreadCancel then
            pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, nil)
        else
            pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nil)
        end else
            pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nil);

    //Let's run the loop and wait for the cancellation request
    repeat
        //Let's change the value of the
        pthread_mutex_lock(ValueMutex);
        inc(ValueCounter);
        pthread_mutex_unlock(ValueMutex);

        sleep(1000);

        //Make the cancellation point

```

```

        pthread_testcancel;
    until false;

    Result := 0;
end;

procedure TfrmThreadDemo.FormCreate(Sender: TObject);
begin
    //Initialize the Mutex
    pthread_mutexattr_init(ValueMutexAttr);
    pthread_mutex_init(ValueMutex, ValueMutexAttr);
end;

procedure TfrmThreadDemo.FormDestroy(Sender: TObject);
begin
    //Let's close down the Mutex
    pthread_mutexattr_destroy(ValueMutexAttr);
    pthread_mutex_destroy(ValueMutex);
end;

procedure TfrmThreadDemo.btnCreateThreadClick(Sender: TObject);
begin
    btnCreateThread.Enabled := false;

    //Define the attributes
    CanCancelThread := cbCanCancel.Checked;
    ImmediateThreadCancel := cbCancelType.ItemIndex = 0;

    //Now let's create the thread
    pthread_create(ThreadID, nil, MyThreadFunction, nil);

    tmrDisplayStatus.Enabled := true;
end;

procedure TfrmThreadDemo.tmrDisplayStatusTimer(Sender: TObject);
begin
    tmrDisplayStatus.Enabled := false;
    try
        pthread_mutex_lock(ValueMutex);
        lblStatusCounter.Caption := inttostr(ValueCounter);
        pthread_mutex_unlock(ValueMutex);
    finally
        tmrDisplayStatus.Enabled := true;
    end;
end;

procedure TfrmThreadDemo.btnCancelThreadClick(Sender: TObject);
begin
    pthread_cancel(ThreadID);
end;

end.

```

pthread_setcanceltype LibC.pas***Syntax***

```
function pthread_setcanceltype(
  CancelType: Integer;
  OldType: PInteger
):Integer;
```

Description

The `pthread_setcanceltype` function defines whether the current thread will immediately cancel when it receives a cancellation request, or if it will wait until it reaches a cancellation point before canceling. Using this function is only effective when the cancel state for the thread is enabled using `pthread_setcancelstate`.

Parameters

CancelType: The `CancelType` parameter defines how the current thread will react to a cancellation request. The thread can be terminated immediately if this parameter is set to `PTHREAD_CANCEL_ASYNCHRONOUS` or the thread will terminate the next time the thread reaches a termination point. A termination point in a thread is anytime the thread calls the `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `sem_wait`, `sigwait`, or `pthread_testcancel` function.

```
const
  PTHREAD_CANCEL_DEFERRED = 0;
  PTHREAD_CANCEL_ASYNCHRONOUS = 1;
```

OldType: This parameter is a pointer to an integer that should hold the value of `CancelType` of the thread before the call to this function was made. If this parameter is `nil`, no value will be returned.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, which would be due to `CancelType` receiving an invalid value, the function will return a valid error code.

See Also

`pthread_setcancelstate`, `pthread_cancel`

Example

See Listing 6.33 - Using the `pthread_setcancelstate` function.

pthread_setschedparam LibC.pas***Syntax***

```
function pthread_setschedparam(
  TargetThreadID: TThreadID;
  Policy: Integer;
  Param: PSchedParam
):Integer;
```


Description

The `pthread_setschedparam` function changes the scheduling policy and priority of the thread specified by the `TargetThreadID` parameter. The possible scheduling policies for the thread are round robin scheduling, scheduling on a first-in/first-out (FIFO) basis, and explicit scheduling. Round robin and FIFO scheduling are only available to processes running as the superuser as these policies are for real-time systems only.

Parameters

TargetThreadID: This parameter is the ThreadID of the thread that will have its scheduling information changed.

Policy: This is the scheduling policy that will be set for the thread. The available options for this parameter are `SCHED_RR` for round robin scheduling, `SCHED_FIFO` for first-in/first-out scheduling, and `SCHED_OTHER` for explicit scheduling. Any value other than these three will result in the function failing and returning an `EINVAL` error code.

```
const
    SCHED_OTHER    = 0;
    SCHED_FIFO     = 1;
    SCHED_RR       = 2;
```

Param: The `Param` parameter is a pointer to a `TSchedParam` structure that contains the new priority value. The value in the `TSchedParam`'s `sched_priority` field must be between the minimum and maximum values for the scheduling policy that is used. The minimum and maximum values for the scheduling policies can be found by using the `sched_get_priority_min` and `sched_get_priority_max` functions.

```
type
    TSchedParam = record
        sched_priority: Integer;
    end;
```

Return Value

If the function is successful, it will return 0. The function will return the `EINVAL` error if the function scheduling policy is not `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, or the priority is not valid for the policy used. If the scheduling is set to `SCHED_FIFO` or `SCHED_RR` and the process is not running as the superuser, the function will return the `EPERM` error code. The function will also return `ESRCH` if the thread has been canceled.

See Also

`pthread_attr_setschedparam`, `pthread_attr_setschedpolicy`, `pthread_getschedparam`

Example

Listing 6.34 - Using the `pthread_setschedparam` function

```
pthread_setschedparam(pthread_self, SCHED_RR,
    sched_get_priority_min(SCHED_RR));
```

pthread_setspecific LibC.pas***Syntax***

```
function pthread_setspecific(
  Key: TPThreadKey;
  Ptr: Pointer
):Integer;
```

Description

In POSIX thread applications, threads can contain data that is specific to the calling thread. This thread-specific data is accessed using a TPThreadKey variable. Unlike a variable, though, thread-specific data needs to be accessed by the interface functions pthread_getspecific and pthread_setspecific.

The pthread_setspecific function will set the value of a thread-specific data key for the current thread. The function will only store a pointer for the thread-specific data that is passed as the Ptr parameter. Any subsequent calls to the pthread_getspecific function will return the pointer passed to this function.

Parameters

Key: This is a TPThreadKey variable that is the thread-specific data key that was created by a call to the pthread_key_create function.

Ptr: This is a pointer to a block of data that will be stored and only accessible from the current thread.

Return Value

If the function is successful, it will return 0. The function will be unsuccessful if the TPThreadKey passed as the Key parameter is invalid and has not been initialized using pthread_key_create. In this case the function will return the EINVAL error code.

See Also

pthread_key_create, pthread_key_delete, pthread_getspecific

Example

See Listing 6.23 - A multi-threaded version library using thread keys.

pthread_sigmask LibC.pas***Syntax***

```
function pthread_sigmask(
  How: Integer;
  const NewMask: PSigSet;
  OldMask: PSigSet
):Integer;
```

Description

The `pthread_sigmask` function will alter the way that signals are accepted on the current thread. The function is similar to the `sigprocmask` function, except that `pthread_sigmask` applies only to the current calling thread instead of the current process. The function allows you to define exactly the signals that the current thread will block, add a signal set to the list of threads that block, and remove a blocking signal set from the current thread.

Parameters

How: The `How` parameter defines if the signal set will be added to the current list. The possible values for this parameter are `SIG_SETMASK`, which means the signal set in the `NewMask` parameter will be blocked from the current thread; `SIG_BLOCK`, which means the signal set in the `NewMask` parameter will be added to the current set of signals that are blocked on the current thread; and `SIG_UNBLOCK`, which means the signal set passed to the `NewMask` parameter will be removed from the set of signals that are blocked on the current thread.

NewMask: This parameter is a pointer to a `TSigSet` signal set. This parameter is used in conjunction with the `How` parameter to either define, add, or remove signals from the current thread's blocking signal set.

OldMask: The `OldMask` parameter is a pointer to a `TSigSet` variable. When this parameter is not `nil`, after this function is called the signal set of the current thread prior to calling this function will be stored in the `TSigSet` that this parameter points to.

Return Value

If the function is successful, it will return 0. The function will fail and return the `EINVAL` error code if the value to the `How` parameter is anything but `SIG_SETMASK`, `SIG_BLOCK`, or `SIG_UNBLOCK`. If either the `NewMask` or `OldMask` parameters point to invalid locations, the function will return the `EFAULT` error code.

See Also

`pthread_kill`, `sigprocmask`, `sigwait`, `kill`, `sigaction`

Example

See Listing 6.24 - Extending the behavior of `TThread`.

pthread_spin_destroy LibC.pas*Syntax*

```
function pthread_spin_destroy(
  var Lock: TPthreadSpinlock
):Integer;
```

Description

The `pthread_spin_destroy` function destroys any resources created for a spin lock as a result of a previous call to the `pthread_spin_init` function. This function should be called when

you are sure no other threads are currently using the spin lock. On the Linux implementation of POSIX threads, this function does nothing and always returns successfully.

Parameters

Lock: This is a valid spin lock variable.

Return Value

This function always returns 0.

See Also

pthread_spin_init, pthread_spin_lock, pthread_spin_trylock, pthread_spin_unlock

pthread_spin_init LibC.pas

Syntax

```
function pthread_spin_init(
  var Lock: TPthreadSpinlock;
  ProcessShared: Integer
):Integer;
```

Description

The pthread_spin_init function sets a TPthreadSpinlock spin lock variable ready to be used within a multi-threaded application. The function will clear the variable so that it appears that the spin lock is not locked and also determines if the spin lock can be used across process boundaries.

Parameters

Lock: This parameter is the spin lock variable to initialize.

ProcessShared: This variable determines whether or not the spin lock can be shared between processes. Under Linux, this parameter is ignored as spin locks can only be shared between processes.

Return Value

This function will always return 0.

See Also

pthread_spin_destroy, pthread_spin_lock, pthread_spin_trylock, pthread_spin_unlock

pthread_spin_lock LibC.pas

Syntax

```
function pthread_spin_lock(
  var Lock: TPthreadSpinlock
):Integer;
```

Description

The `pthread_spin_lock` function will lock a spin lock to protect a section of code in a similar way a mutex would protect a section of code. The spin lock is different from the mutex as it is implemented to be the fastest possible method of protecting a resource.

Parameters

Lock: This is the spin lock variable that will be locked.

Return Value

This function will always return 0.

See Also

`pthread_spin_init`, `pthread_spin_destroy`, `pthread_spin_trylock`, `pthread_spin_unlock`

pthread_spin_trylock LibC.pas*Syntax*

```
function pthread_spin_trylock(
  var Lock: TThreadSpinlock
):Integer;
```

Description

The `pthread_spin_trylock` function attempts to lock a spin lock in the same way the `pthread_spin_lock` function does, but instead of the function blocking until the spin lock can be taken, the function will return an error code to state that you cannot obtain an immediate lock.

Parameters

Lock: This is the spin lock variable that will be locked.

Return Value

If the lock on the spin lock can be immediately obtained, the function will return 0. If the lock cannot be immediately obtained, the function will return the EBUSY error code.

See Also

`pthread_spin_init`, `pthread_spin_destroy`, `pthread_spin_lock`, `pthread_spin_unlock`

pthread_spin_unlock LibC.pas*Syntax*

```
function pthread_spin_unlock(
  var Lock: TThreadSpinlock
):Integer;
```

Description

The `pthread_spin_trylock` function releases a lock on a spin lock previously obtained by a call to `pthread_spin_lock` or `pthread_spin_unlock`. After the spin lock has been unlocked, other threads will have the opportunity to gain the lock on the spin lock.

Parameters

Lock: This is the spin lock variable that will be unlocked.

Return Value

This function will always return 0.

See Also

`pthread_spin_init`, `pthread_spin_destroy`, `pthread_spin_lock`, `pthread_spin_trylock`

pthread_testcancel* LibC.pas*Syntax**

```
procedure pthread_testcancel;
```

Description

The `pthread_testcancel` function is used to see if the termination of the current thread has been requested, by using the `pthread_cancel` function or some other mechanism. If the thread has been canceled, this function will terminate the thread. If another thread is waiting on this function using the `pthread_join` function, the data returned will return as if the call to `pthread_exit(PTHREAD_CANCELED)` was made.

```
const
  PTHREAD_CANCELED = Pointer(-1);
```

See Also

`pthread_cancel`, `pthread_setcancelstate`, `pthread_setcanceltype`

Example

See Listing 6.10 - A Web server using thread pooling.

pthread_yield* LibC.pas*Syntax**

```
function pthread_yield:Integer
```

Description

The `pthread_yield` function allows other threads to process without causing the current thread to block. As POSIX threads are implemented on Linux using processes, internally this function calls `sched_yield` to allow other processes the ability to run.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return a valid error code.

sem_close LibC.pas**Syntax**

```
function sem_close(  
  var __sem: TSemaphore  
):Integer;
```

Description

Theoretically, the `sem_close` function will perform any cleanup operations on a named semaphore when the semaphore is no longer required. As named semaphores are not supported on Linux's implementation of POSIX threads, calling this function will always result in the `ENOSYS` error being returned.

Return Value

As this function is not implemented, this function will always return `-1` and a call to the `errno` function will return the `ENOSYS` error code.

See Also

`sem_init`, `sem_destroy`, `sem_open`

sem_destroy LibC.pas**Syntax**

```
function sem_destroy(  
  var __sem: TSemaphore  
):Integer;
```

Description

The `sem_destroy` function will perform any cleanup operations on an unnamed semaphore when the semaphore is no longer required. The semaphore passed into the function should be an unnamed semaphore that was initialized by a call to the `sem_init` function. Named semaphores created with the `sem_open` function should use the `sem_close` function to manage the cleanup process of the semaphore.

Parameters

`__sem`: This parameter is the unnamed semaphore that was initialized using the `sem_init` function.

Return Value

If the function is successful, it will return 0. If, however, the semaphore is locked, the function will return `-1` and a call to the `errno` function will result in the `EBUSY` error code being returned.

See Also

`sem_init`, `sem_post`, `sem_wait`, `sem_trywait`, `sem_timedwait`

`sem_getvalue` LibC.pas**Syntax**

```
function sem_getvalue(
  var __sem: TSemaphore;
  var __sval: Integer
):Integer;
```

Description

The `sem_getvalue` function will return the number of resources that are currently available in the semaphore. This function will not really add much functionality to a multi-threaded application, but it is useful in debugging an application to determine the current status of a semaphore protecting some resource pool. The value returned in the `__sval` parameter will indicate how many resources are available to lock. A value of 0 will indicate that there are no resources left and that the semaphore is locked, while a value greater than 0 indicates that the thread is unlocked. This function can be used on named or unnamed POSIX semaphores.

Parameters

`__sem`: This is the semaphore variable that will be examined.

`__sval`: After the function is called, this variable parameter will contain the number of resources the semaphore has available to it. A positive value will indicate that the semaphore is unlocked, while 0 will indicate that the semaphore is locked.

Return Value

This function will always return 0.

See Also

`sem_init`, `sem_post`, `sem_wait`, `sem_trywait`, `sem_timedwait`

Example

See Listing 6.10 - A Web server using thread pooling.

`sem_init` LibC.pas**Syntax**

```
function sem_init(
  var __sem: TSemaphore;
  __pshared: LongBool;
  __value: LongWord
):Integer;
```


Description

The `sem_init` function will initialize an unnamed semaphore. This function also defines the number of resources the semaphore is protecting in the `__value` parameter, as well as if the semaphore can be shared between processes. As this semaphore is initialized by a call to the `sem_init` function, the semaphore must be released by a call to the `sem_destroy` function.

Parameters

`__sem`: The `__sem` parameter is a `TSemaphore` variable that will be initialized.

`__pshared`: This parameter determines if the semaphore can be shared between processes. If this value is set to `PTHREAD_PROCESS_SHARED`, the semaphore can be shared between processes. If this parameter is `PTHREAD_PROCESS_PRIVATE`, the semaphore cannot be shared between processes. Although the POSIX standard mentions that semaphores can be shared between processes, the Linux implementation will not allow this. Stating that you want the semaphore to be shared will result in the function failing and the `ENOSYS` error code being returned by a call to the `errno` function.

`__value`: The `__value` parameter specifies the number of resources that the semaphore will be initially set up to protect. This parameter must be positive and less than the constant value `SEM_VALUE_MAX`. Otherwise, the function will result in an error and the `errno` function will return the `EINVAL` error code.

Return Value

If the function is successful, it will return 0. If the function is unsuccessful, it will return -1.

See Also

`sem_destroy`, `sem_open`, `sem_close`

Example

See Listing 6.10 - A Web server using thread pooling.

sem_open LibC.pas*Syntax*

```
function sem_open(
  __name: PChar;
  __oflag: Integer
):PSemaphore;
```

Description

According to the POSIX standard, the `sem_open` function should create a named semaphore with the name specified in the `__name` parameter. The `__oflag` parameter specifies the creation parameters similar to the way a file descriptor is opened. Because named semaphores are not used in Linux's implementation of POSIX threads, calling this function will

always fail; the function will return `-1` and a call to the `errno` function will return the `ENOSYS` error code.

Return Value

As this function is not implemented, it will always return `-1` and a call to the `errno` function will return the `ENOSYS` error code.

See Also

`sem_close`, `sem_init`, `sem_destroy`

sem_post **LibC.pas**

Syntax

```
function sem_post(  
  var __sem: TSemaphore  
):Integer;
```

Description

The purpose of the `sem_post` function is to reissue a resource to a semaphore so that the resource can be protected once again by the semaphore. The semaphore will always contain a value that represents the number of resources it is protecting. Calling this function simply increases the number of resources being protected. As such, other threads waiting for a resource to become available will have the opportunity to obtain this resource.

Parameters

`__sem`: The `__sem` parameter is a valid `TSemaphore` variable that was previously waited on using `sem_wait`, `sem_timedwait`, or `sem_trywait`.

Return Value

If the function is successful and the resource can be successfully placed back into the management of the semaphore, the function will return `0`. There are several reasons why the function may fail. If the new value of the semaphore is larger than the allowable value, which is defined by the `SEM_VALUE_MAX` constant, the function will return `-1` and a call to the `errno` function will return `ERANGE`. In the unlikely event that the thread library could not initialize, the function will return `-1` and the `errno` function will return the `EAGAIN` error code.

See Also

`sem_wait`, `sem_trywait`, `sem_timedwait`

Example

See Listing 6.10 - A Web server using thread pooling.

sem_timedwait LibC.pas**Syntax**

```
function sem_timedwait(
  var __sem: TSemaphore;
  const __abstime: timespec
):Integer;
```

Description

The `sem_timedwait` function behaves in a similar manner to the `sem_wait` function, with the exception that `sem_timedwait` only allows a specified period of time to pass to obtain a lock on a semaphore; otherwise, the function will result in an error. The time that the current thread will wait is defined by the `__abstime` parameter.

Parameters

`__sem`: This parameter is a valid `TSemaphore` variable.

`__abstime`: The `__abstime` parameter is a `timespec` variable that defines the maximum time the thread should wait to attempt to obtain a lock on the semaphore. If more than 1,000,000,000 seconds are placed into this time parameter, the function will return the `EINVAL` error code.

Return Value

If the function is successful, it will return 0. If more than 1,000,000,000 seconds are placed into the time parameter, the function will return the `EINVAL` error code. If the timeout period was reached, the function will return the `ETIMEDOUT` error code.

See Also

`sem_wait`, `sem_trywait`, `sem_post`

sem_trywait LibC.pas**Syntax**

```
function sem_trywait(
  var __sem: TSemaphore
):Integer;
```

Description

The `sem_trywait` function is similar to the `sem_wait` function, in that the function attempts to obtain a lock on a resource protected by a semaphore. The only difference between this function and the `sem_wait` function is that this function will return immediately if the lock cannot be obtained. The function will return `-1` and the `EAGAIN` error code will be returned from a call to the `errno` function.

Parameters

`__sem`: This parameter is a valid `TSemaphore` variable.

Return Value

If the lock can be obtained immediately, this function will return 0 and the value of the semaphore will be decreased by 1. If the lock cannot be obtained immediately, the function will return -1 and a call to the `errno` function will return the EAGAIN error code.

See Also

`sem_wait`, `sem_timedwait`, `sem_post`

`sem_unlink` *LibC.pas*

Syntax

```
function sem_unlink(
  __name: PChar
):Integer;
```

Description

The purpose of the `sem_unlink` function is to remove a named semaphore from the operating system. In most cases, named semaphores will be stored on the file system and this function is used to manage the cleanup of them. As named semaphores are not implemented in LinuxThreads, this function will always fail by returning -1 and the ENOSYS error code will be returned by a call to the `errno` function.

Return Value

On Linux, this function will always return -1, and a call to the `errno` function will return the ENOSYS error code.

`sem_wait` *LibC.pas*

Syntax

```
function sem_wait(
  var __sem: TSemaphore
):Integer;
```

Description

The `sem_wait` function attempts to obtain a single lock on a semaphore resource. Internally, the semaphore will have a value that represents the number of resources it has to protect. The `sem_wait` function, in most cases, takes a resource from the semaphore. If the semaphore is positive, i.e., it has resources to allocate, the semaphore's value will be decreased by 1. If, however, the value of the semaphore is 0, then the function will block until a resource is made available by another thread calling the `sem_post` function on the semaphore.

Parameters

`__sem`: This parameter is a valid `TSemaphore` variable.

Return Value

Under LinuxThreads, this function is always successful, which means the function will return 0 and the value of the semaphore will be decreased by 1.

See Also

`sem_timedwait`, `sem_trywait`, `sem_post`

Example

See Listing 6.10 - A Web server using thread pooling.

Conclusion

In this chapter you have learned how POSIX threads work under Linux and how to exploit them to your advantage. You have also learned about the various synchronization techniques at your disposal, such as mutexes, semaphores, condition variables, spin locks, read/write locks, and barriers.

As most multi-threaded applications you will build with Kylix will be using TThread instead of using the pthread functions, you will really only need this chapter when you are managing synchronization, although it does help to know what is going on under the hood. Writing multi-threaded applications is difficult. For many of you, even writing single-threaded applications may be a challenge. By understanding the inner workings of threads under Linux, you can better prepare yourself for when your software needs to pat its head and rub its stomach at the same time.

The Socket API Functions

Introduction

Now that you've seen the various interprocess methods on Linux, we will look at probably the most widely used computer-based communication method on the planet today, known as sockets. The previous interprocess communication methods that have been discussed so far only allow interprocess communication within the same machine. Sockets go outside machine boundaries and allow communication with other processes running on different servers in the next room, across the street, or possibly on the other side of the world.

There are many facets to understanding sockets, and this chapter provides you with a firm grounding on using sockets within your application, without going into techniques that you are unlikely to need. Working with sockets can be sometimes complicated, and entire books are devoted to programming with sockets alone.

As a result, this chapter will introduce you to the fundamentals of working with the socket functions in the Linux API. By the end of the chapter you should be familiar with the concept of Internet addressing, programming sockets, making socket client and server applications, and how you can integrate them with the existing socket components that are already in Kylix.

What is a Socket?

The first question that you are probably asking yourself is, what is a socket? A socket is a communication mechanism that allows communication of data from a client process to a server process. These processes may or may not be on the same machine or running the same operating system, as this communication mechanism normally takes place over a network.

A socket is similar in concept to a telephone switch in that a particular port on the telephone switch of the caller is connected to a particular port on another or the same telephone switch, depending on if the phone call was a local call or not.

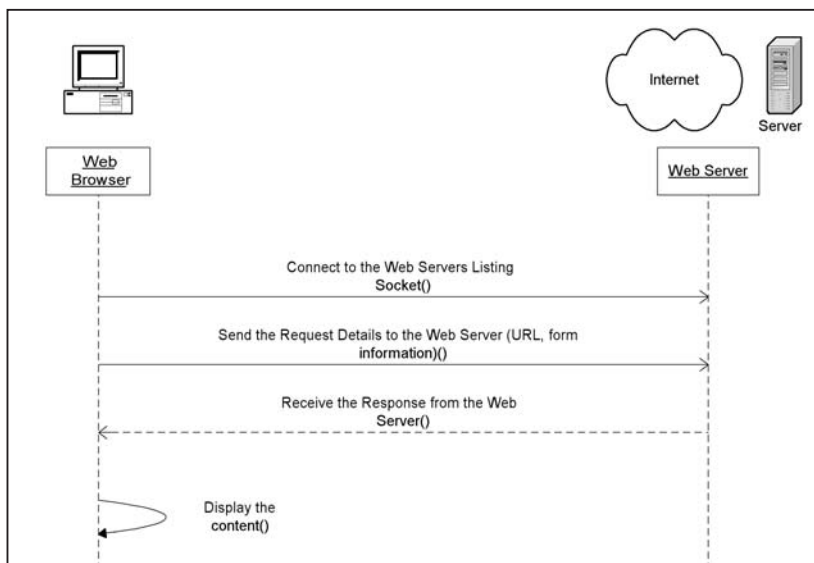
Although this communication occurs over telephone lines, it is not too different from what happens with socket communication on a network.

In the telephone example, a caller wants to place a call to a specific area code and phone number. The phone system then attempts to connect to the phone number the caller wants and then connects the caller so that the caller can now communicate with the other phone system.

Using sockets on a network is a similar scenario. The caller process would know which machine and port number it wants to connect to. It would then use the underlying network transport layer to attempt to connect to the port on the server machine, and if a connection could be established then communication could occur between the systems.

Because sockets can communicate between machines, they are the foundation of today's Internet communication. The popular Internet services you are familiar with, such as Web servers, use sockets to handle their communication. In fact, each time you open a Web browser you are connecting your socket client (Web browser) to the socket server (Web server), and the socket client is requesting the information from the server. Web servers are not the only protocol that can be used. Familiar Internet services, such as File Transfer Protocol (FTP), and e-mail systems, such as the Simple Mail Transfer Protocol (SMTP) and Post Office Protocol (POP), all use sockets as their underlying communication methods. What makes each individual Internet service different is the standard for accessing and retrieving information.

Figure 7.1 - Diagram of socket communication between a browser and Web server



Steps in Establishing a Socket Connection

Before going on, you should examine the steps involved in creating a connection to a socket for a client process. We will deal with server sockets in a later section, but for now the process is quite straightforward.

1. Create a socket using the desired network protocol for the communication.
2. Define the address information for the machine you want to connect to. This includes the machine address, the port number, and the network protocol being used for the connection.
3. Attempt a connection to the server.
4. By this stage you will either not be able to communicate with the server or you will have an active socket on which you can read or write information.

In the following sections we will do the four steps listed above to create a simple date/time client socket application. Implementing a date/time client is one of the “Hello, World” equivalent applications for socket communication. In this application, you connect to the server on port 13 and bring back all the content, which will be a date in a human-readable format such as Tuesday, May 01, 2001 02:26:52-EST. So, the first thing you need to do is create the socket.

Creating a Socket

It goes without saying that before you can have socket communication, you need a socket. Luckily, creating a socket is quite a simple. To create a socket, call the `socket` function.

```
type
    TSocket = TFileDescriptor;

function socket(__domain, __type, __protocol: Integer): TSocket;
```

The `socket` function takes three parameters: the domain of the socket, the type of the socket, and the protocol used.

The domain refers to the address family that is to be used. The address family defines what address format is used when defining which machine to communicate with. The address family also defines the underlying network transport to use. Some of the types of domains include `AF_UNSPEC`, `AF_LOCAL`, `AF_UNIX`, `AF_FILE`, `AF_INET`, `AF_AX25`, `AF_IPX`, `AF_APPLETALK`, `AF_NETROM`, `AF_BRIDGE`, `AF_ATMPVC`, `AF_X25`, `AF_INET6`, `AF_ROSE`, `AF_DECnet`, `AF_NETBEUI`, `AF_SECURITY`, `AF_KEY`, `AF_NETLINK`, `AF_ROUTE`, `AF_PACKET`, `AF_ASH`, `AF_ECONET`, `AF_ATMSVC`, `AF_SNA`, `AF_IRDA`, `AF_PPPOX`, and `AF_MAX`.

Although there are many address families to use, there are really only a few that you will work with. The main family you will use and the only address family covered in this chapter is `AF_INET`, which is Internet Protocol version 4 (IPv4), also referred to as IP. This protocol is the one that is used on the Internet today. You have probably seen an IP address in the form of 4-byte values separated by periods like 192.168.22.121.

There are other address families that can be useful. The `AF_UNIX` address family is used only locally on the machine. Another address family is the newest version of the Internet Protocol, IPv6, which is the address family `AF_INET6`. Because the world is running short on IPv4 addresses, the new IPv6 format was created and will undoubtedly be used more in the forthcoming years. An IPv6 address is in the form of 128 bits, and is represented by hexadecimal values such as 4A3F:AE67:F240:56C4:3409:AE52:440F:1403. IPv6 will be implemented as a replacement for IPv4 and will be the protocol of choice for the Internet in the future, but for the moment, the Internet is currently running using IPv4.

The next parameter to the `socket` function is the `__type` option. This defines the communication type of the socket and how communication is to be sent. There are three realistic options you can use within your applications today. These are `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

The `SOCK_STREAM` option means that the communication will use Transmission Control Protocol (TCP), which provides a reliable connection-based method of sending data across the socket. The term “connection-based” refers to establishing a connection and

keeping the connection active. You would use TCP as your communication type when you need to ensure that the data is communicated.

On the other hand, the `SOCK_DGRAM` option uses an unreliable connectionless option of sending data. `SOCK_DGRAM` represents the User Datagram Protocol (UDP). Rather than establishing a connection, UDP uses broadcasts to send messages to the listening server. The problem with this approach is that the server may not actually get the message, and there is no way to make sure that the message arrived. Normally, you would only use UDP when you need to send data that you do not really need confirmation for.

The last of the options is using raw sockets (`SOCK_RAW`). This allows you to read and write low-level network information over the socket. With raw sockets you can create applications such as network monitor activity, ping other machines to see if the machine is active, and perform many other low-level functions. Raw sockets are beyond the scope of this book; I will mainly cover Transmission Control Protocol with some examples of User Datagram Protocol.

The last of the parameters of the socket function is the protocol parameter. This parameter defines which network protocol to use for the socket, even though there may be several possible protocols to use, such as running older network protocols NetBeui over IP, which can occur. In most cases, however, there is only one protocol, depending on the address family that is used in the `__domain` parameter, so in 99 percent of cases this value is 0, which indicates that the socket should use the default protocol for the address family.

When the parameters are passed in, a successful call to the socket function will result in a valid file descriptor being returned as shown in Listing 7.1. These are the same file descriptors that were used in Chapter 3 for reading and writing to files. This is due to Linux's notion that everything should be treated like a file. When we connect to the socket, you will see that reading and writing to the socket are done in exactly the same way that is done with a file.

Listing 7.1 - Creating a socket

```
var
    SocketFileDescriptor: integer;
...
begin
    ...
    //Create the socket
    SocketFileDescriptor := socket(AF_INET, SOCK_STREAM, 0);
    if SocketFileDescriptor = -1 then
        raise Exception.Create('Could not create the socket');
```

Now that you have the socket, the next step is to connect the socket to a socket server. To do that, we need to know the location of the server in question.

Socket Addresses

Address information is extremely important to know when working with sockets. You saw in the section on socket definition that the entire communication process depended on knowing which machine and port to connect to so that communication can occur. Defining the address of the server your socket is connecting to is done with a `TSockAddr` data type.

The `TSockAddr` data type holds the address information when you are connecting to a server. `TSockAddr` is actually a record with unions and is declared as:

```
type
  sockaddr = {packed} record
    case Integer of
      0: (sa_family: sa_family_t;
          sa_data: packed array[0..13] of Byte);
      1: (sin_family: sa_family_t;
          sin_port: u_short;
          sin_addr: TInAddr;
          sin_zero: packed array[0..7] of Byte);
    end;
  TSockAddr = sockaddr;
```

This data type holds the information about what server to connect to and which port to connect on. Although the data type looks a little daunting, there are a few simple rules to remember and many useful functions that allow you to define an address.

First, the `TSockAddr` data type defines a union, so that an address is represented by the address family and then the address details. When using Internet IPv4 sockets the family will be `AF_INET`, which is the same value that was passed to the socket function earlier. The other information is normally the port that you wish to connect to and the IPv4 address of the server.

You may be thinking that the value for the port you are connecting to in the `sin_port` field would be relatively straightforward and likewise for the address; you just place the values in for the port and the IP address and away you go. Unfortunately, this is not the case. Numbers are stored internally on one system in a different way than they are stored on another. A 16-bit word value, which is used for a port number, takes two bytes. On one system the value that represents the high-order byte is stored first and the lower byte value stored second. On other systems, the lower value may be stored first, followed by the high-order value. To make sure that all networks use the same byte order, all values used in sockets are actually written in binary using network byte ordering.

There are several functions available to the developer to allow you to change a number value to use network byte ordering. The functions to manage network byte ordering are shown in Listing 7.2.

Listing 7.2 - Function definitions for converting values using network byte ordering

```
function ntohl(__netlong: uint32_t): uint32_t; cdecl;
function ntohs(__netshort: uint16_t): uint16_t; cdecl;
function htonl(__hostlong: uint32_t): uint32_t; cdecl;
function htons(__hostshort: uint16_t): uint16_t; cdecl;
```

The main functions you will use to convert values typically convert values that are valid on the host system into network byte ordered values, which are suitable for use on the Internet. The `htonl` and `htons` functions do these operations by converting values that are valid for the ordering process of the host machine to a valid network byte value that is acceptable for use on the network.

The `ntohl` function will convert a network byte order value to the unsigned integer value that is valid for the host machine. Likewise, the `ntohs` function will convert a network byte-ordered word value to a word value that is valid for the host.

So, the port that will be used for the DayTime socket client is port 13. To make sure that we connect with the right value we pass `htons(13)` to the value of the `sin_port` field of our `TSockAddr` when connecting.

The next part of defining a socket address is specifying the particular computer that will be able to communicate with another system. As a result, when sockets are used to communicate they need to have a way of identifying the machine they wish to communicate with. This is done by assigning each machine one or multiple unique addresses.

How the address is defined is dependent on which protocol the socket is using for transport. The protocol most often used is Internet Protocol version 4 (IPv4), the one used on the Internet today. Most communication on the Internet uses the Transmission Control Protocol (TCP), which uses the Internet Protocol (IP) for communication, which is why you have probably heard the term “TCP/IP.”

Constructing a value for the `sin_addr` field of the `TSockAddr` is normally done by passing the result of a call to the `inet_addr` function. The `inet_addr` function shown in Listing 7.3 is used to convert a string containing the IP address and return a valid address.

Listing 7.3 - The `inet_addr` function

```
function inet_addr(__cp: PChar): in_addr_t; cdecl;
```

So to specify a particular IP address that you wish to connect to, you would simply pass the IP string into the `inet_addr` function and you have your address value. Putting this together will give you something similar to Listing 7.4.

Listing 7.4 - Defining an address for use on the network

```
var
  SocketAddress: TSockAddr;
  ...
begin
  //Define the address of where I am connecting to
  with SocketAddress do
  begin
    sin_family := AF_INET;
    //Convert the string into an IPv4 Address.
    sin_addr.S_addr := inet_addr(PChar(ServerIPAddress));
    sin_port := htons(PortNo);
  end;
  ...
```

Connecting a Socket Client to a Server

Now that you have the address information on the server that you want to connect to, it is just a matter of connecting to the server. The function for connecting to the socket server is the `connect` function, which is shown in Listing 7.5.

Listing 7.5 - The `connect` function

```
function connect(__fd: TSocket; const __addr: sockaddr; __len: socklen_t):
  Integer; cdecl;
```

The connect function takes three parameters: the file descriptor of the socket that was created earlier, the address of the server you will be connecting to, and the size of the record structure that holds the address information.

The `__fd` parameter requires the resulting file descriptor that was returned from a successful call to the socket function shown in Listing 7.1. The `__addr` parameter takes the address information that defined the connection details to the server and the `__len` parameter will simply take the data size of the `TSockAddr` record.

When the call to connect is successful, you have your official socket connection using TCP/IP and you can read and write from the socket file descriptor just as you would a file descriptor on a file. Once you have finished your communication with the socket, simply use the `__close` function to close the socket file descriptor. All the steps to create the socket client application are demonstrated in Listing 7.6. You can see that it creates the socket, defines the address it wants to connect to, connects, and then reads and writes the appropriate data.

Listing 7.6 - Code to connect to a time server

```
unit unitGetTimeForm;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

type
  TfrmSimpleTimeClient = class(TForm)
    Label1: TLabel;
    lblCurrentTime: TLabel;
    btnGetTime: TButton;
    procedure btnGetTimeClick(Sender: TObject);
  public
    { Public declarations }
  end;

var
  frmSimpleTimeClient: TfrmSimpleTimeClient;

implementation

{$R *.xfm}

uses
  Libc;

type
  ETimerSocketServerException = class(Exception);

function GetTimeFromServer(ServerIPAddress: string = '127.0.0.1'; PortNo: Word = 13):
string;
var
  SocketAddress: TSockAddr;
  SocketFileDescriptor: integer;
  DataContent: string;
```

```

    DataBuffer: array[0..1023] of char;
    DataRead: integer;
begin
    //This function connects to the server and then reads all the
    //content from a time standard.
    DataContent := '';

    //Create the socket
    SocketFileDescriptor := socket(AF_INET, SOCK_STREAM, 0);
    if SocketFileDescriptor = -1 then
        raise ETimerSocketServerException.Create('Could not create the socket');

    //Define the address of where I am connecting to
    with SocketAddress do
    begin
        sin_family := AF_INET;
        //Convert the string into an IPv4 Address.
        sin_addr.S_addr := inet_addr(PChar(ServerIPAddress));
        sin_port := htons(PortNo);
    end;

    //Attempt connection
    if connect(SocketFileDescriptor, SocketAddress, sizeof(SocketAddress)) = -1 then
    begin
        //We could not connect to the socket. We will close the socket and get out of here
        __close(SocketFileDescriptor);

        raise ETimerSocketServerException.Create('Could not connect to the server socket');
    end;

    //Read all the data from the server
    DataRead := __read(SocketFileDescriptor, DataBuffer, 1024);
    while DataRead > 0 do
    begin
        DataContent := DataContent + Copy(StrPas(DataBuffer), 1, DataRead);
        DataRead := __read(SocketFileDescriptor, DataBuffer, 1024);
    end;

    //Remove the socket
    __close(SocketFileDescriptor);

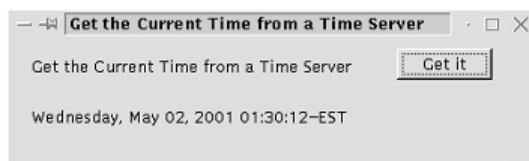
    Result := DataContent;
end;

procedure TfrmSimpleTimeClient.btnGetTimeClick(Sender: TObject);
begin
    lblCurrentTime.Caption := GetTimeFromServer('127.0.0.1', 13);
end;

end.

```

Figure 7.2 -
The simple
DayTime
socket client



Having a Conversation with a Socket Server

The previous section has provided the basic skills to create a simple socket client. The problem with the time server example is that the data is only read from the server. Normally when you connect to the server, you make a request for some data and the data is sent back, or you may have many questions and responses within the same connection. This is the nature of most Internet protocols such as HTTP, finger, FTP, and many others.

The finger protocol is another simple protocol that allows you to query a server to gather information on a particular user or information about who is currently logged onto a particular domain. The information request is done the same way that was done with the DayTime server — create the socket, define the address we are connecting to, and connect. The finger protocol requests a string with the query information in it. The query will be in the format ‘someuser’ (without the quotes) to gather information for a particular user or a blank string to retrieve a list of logged-on users.

Figure 7.3 - Steps to finger a particular user/domain

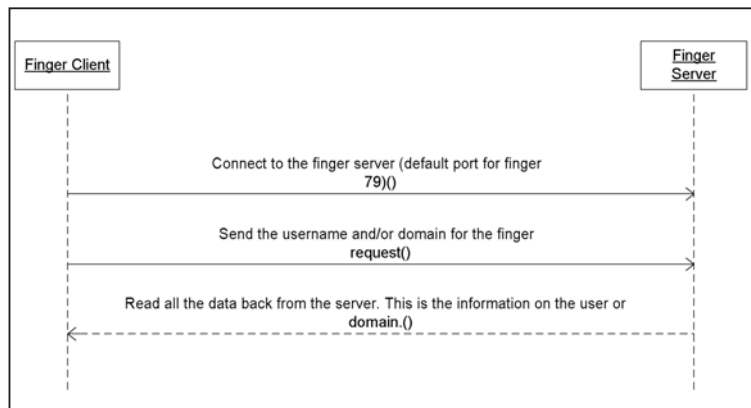


Figure 7.3 demonstrates the process for a simple conversation to request finger information from a particular server. This is, again, a simple communication process, as many of the Internet protocols are. Generally, when you are communicating with a server process there is a normal amount of reading and writing of data in the same way a conversation occurs between two people.

The difference between the finger client and the DayTime client is that this client demonstrates connecting to another server when only the host name of the server name was known. The host name is the user-friendly name that refers to a particular IP address. Host names are stored in a Domain Name System (DNS) server and each machine is typically configured with the location details of a DNS server that it can access to obtain the IP address when the only information that is known is the hostname, such as `www.linux.org`.

Listing 7.7 - Code to obtain finger details from a server

```

unit unitFingerDataForm;

interface

uses

```

```
SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms,
QDialogs, QStdCtrls, QExtCtrls, Libc;
```

```
type
```

```
TfrmFingerClient = class(TForm)
    mmFingerInfo: TMemo;
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    edUsername: TEdit;
    edDomain: TEdit;
    Label3: TLabel;
    btnFinger: TButton;
    btnClose: TButton;
    procedure btnCloseClick(Sender: TObject);
    procedure btnFingerClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

```
var
```

```
    frmFingerClient: TfrmFingerClient;
```

```
implementation
```

```
{ $R *.xfrm }
```

```
type
```

```
    EFingerSocketServerException = class(Exception);
```

```
function HostNameToIPAddress(hostname: string): string;
```

```
var
```

```
    HostEntry: PHostEnt;
```

```
begin
```

```
    //This function converts a hostname into a valid IP address
```

```
    if hostname <> '' then
```

```
    begin
```

```
        //Check for an IP address being passed in
```

```
        if isdigit(ord(hostname[1])) = 1 then
```

```
            Result := hostname
```

```
        else begin
```

```
            HostEntry := gethostbyname(PChar(hostname));
```

```
            if HostEntry <> nil then
```

```
                with HostEntry^ do
```

```
                    Result := Format('%d.%d.%d.%d', [ord(h_addr^[0]), ord(h_addr^[1]),
                    ord(h_addr^[2]), ord(h_addr^[3])]);
```

```
            end;
```

```
        end else
```

```
            raise Exception.Create('The host name was not valid.');
```

```
    end;
```

```
function FingerUserDomain(User: string;
```

```
    FingerServerHostNameOrAddress: string = '127.0.0.1'; PortNo: Word = 79): string;
```

```

var
  SocketAddress: TSocketAddr;
  SocketFileDescriptor: integer;
  DataContent: string;
  DataBuffer: array[0..1023] of char;
  DataRead: integer;
  FingerInfo: array[0..200] of char;
  RealIPAddress: string[20];
  pRealIP: array[0..20] of char;
begin
  //This function connects to the server and then reads all the
  //content from a time standard.
  DataContent := '';

  //Create the socket
  SocketFileDescriptor := socket(AF_INET, SOCK_STREAM, 0);
  if SocketFileDescriptor = -1 then
    raise EFingerSocketServerException.Create('Could not create the socket');

  //Define the address of where I am connecting to
  with SocketAddress do
    begin
      sin_family := AF_INET;
      //Convert the string into an IPv4 Address.
      RealIPAddress := HostNameToIPAddress(FingerServerHostNameOrAddress);
      StrPCopy(pRealIP, RealIPAddress);
      sin_addr.S_addr := inet_addr(pRealIP);
      sin_port := htons(PortNo);
    end;

  //Attempt connection
  if connect(SocketFileDescriptor, SocketAddress, sizeof(SocketAddress)) = -1 then
    begin
      //We could not connect to the socket. We will close the socket and get out of here
      __close(SocketFileDescriptor);

      raise EFingerSocketServerException.Create('Could not connect to the server
      socket');
    end;

  //Here we will send the data to the server. This will be the username (optional) and
  // the domain in the format username@domain.
  StrPCopy(FingerInfo, Trim(User)+#10);
  __write(SocketFileDescriptor, FingerInfo, StrLen(FingerInfo)+1);

  //Read all the data from the server
  DataRead := __read(SocketFileDescriptor, DataBuffer, 1024);
  while DataRead > 0 do
    begin
      DataContent := DataContent + Copy(StrPas(DataBuffer), 1, DataRead);
      DataRead := __read(SocketFileDescriptor, DataBuffer, 1024);
    end;

  //Remove the socket
  __close(SocketFileDescriptor);

```



```

    Result := DataContent;
end;

procedure TfrmFingerClient.btnFingerClick(Sender: TObject);
var
    IPToConnectTo: string;
begin
    //If no domain is specified, then connect locally
    if edDomain.Text = '' then
        IPToConnectTo := '127.0.0.1'
    else
        IPToConnectTo := edDomain.Text;

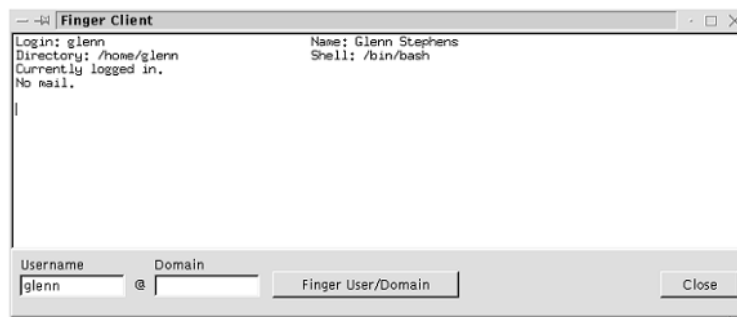
    mmFingerInfo.Lines.Text := FingerUserDomain(edUserName.Text, IPToConnectTo);
end;

procedure TfrmFingerClient.btnCloseClick(Sender: TObject);
begin
    Close;
end;

end.

```

*Figure 7.4 -
The Finger
Client
application*

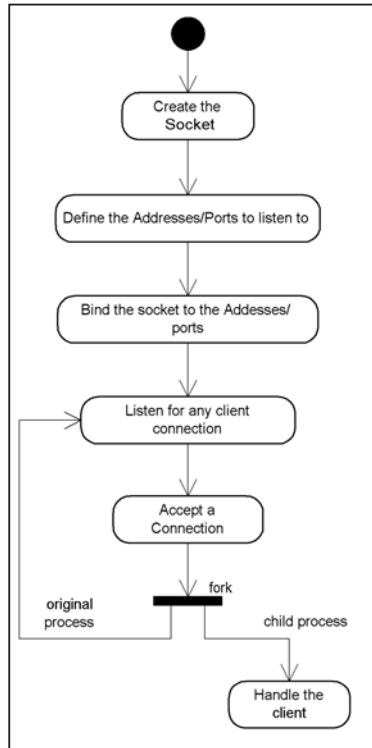


Creating a Socket Server

Now that you have learned to create a simple socket client, the next step is to create a socket server. The process of creating a socket server is not much different from creating a socket client. In fact, many of the steps to create a client socket are the same ones to create a server socket.

The steps involved when creating a socket server are shown in Figure 7.5. The basic steps are to create a socket, define the address that the socket will listen to, bind the socket to that particular port, listen for new connections from client sockets, and accept the connections when the connections arrive.

Figure 7.5 -
Diagram for
creating a
socket server



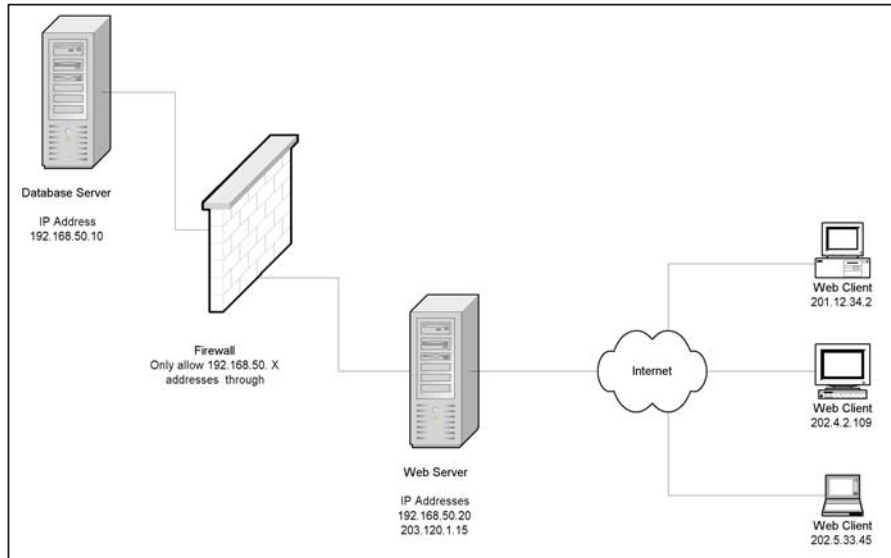
Now that we have a DayTime client, let's make a DayTime server to demonstrate the process of receiving information from the client.

Creating a Socket and Listening Address for a Server Socket

Creating a socket for a server is done in the same manner that it is done for a client application. In fact, the same parameters for the socket function are used to create the socket.

Creating an address for a server to listen to is a different story, however. When you create a client socket, you know the address details of the server that you want to listen to. When you have a server socket, the machine that the server is running on may have multiple IP addresses configured to the machine as shown in Figure 7.6. This will then give you the option to listen either on a specific IPv4 address or to all the IPv4 addresses. In some cases you will not listen on all IP addresses as each IP address may be used for a specific need. In Figure 7.6, you see a Web server with multiple IP addresses, with one IP address a connection to the Internet (203.120.1.15) and the other IP used to connect to a local database server (192.168.50.20). This demonstrates the need to listen on only one IP address.

Figure 7.6 -
A machine
with multiple
IP addresses
configured



In some cases you will want to allow connection on a specific IP address. In Listing 7.8, the code demonstrates defining a particular address for the server to listen on. This is the same way you define an address for a client socket so you should already be familiar with this technique.

On the other hand, you may want your server to listen for a service on all available IP addresses to handle connections. To set up your server to listen on all IP addresses, you would use the technique in Listing 7.9 where the `INADDR_ANY` constant is used to tell the socket to listen on all IP addresses. This will allow any socket connection to that port on that machine to be handled by your server.

Listing 7.8 - Defining a specific address for a client socket to connect to

```

var
  //Server variables
  ServerSocketID: integer;
  ServerAddressToBindTo: TsockAddrIn;
begin
  ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);
  try
    //Define the address to connect to
    with ServerAddressToBindTo do
      begin
        sin_family := AF_INET;
        //listen on a particular IP address
        sin_addr.S_addr := inet_addr('203.92.12.34');
        //listen on port 13
        sin_port := htons(13);
      end;
  end;
  ...

```

Listing 7.9 - Defining an address so all available IP addresses will be used

```

var
    //Server variables
    ServerSocketID: integer;
    ServerAddressToBindTo: TsockAddrIn;

begin
    ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);

    //Define the address to connect to
    with ServerAddressToBindTo do
    begin
        sin_family := AF_INET;
        //listen to any IP address
        sin_addr.S_addr := htonl(INADDR_ANY);
        //listen on port 13
        sin_port := htons(13);
    end;
    ...

```

One nasty little bit of information to know is that when you create an application that listens for a particular server on a port number less than 1,024 under Linux, the process that is running the connection must be running as the superuser, which means you will need to change the owner of the server application using the Linux `chown` command. Changing the owner to the superuser can present a potential security issue if your server application does things such as executing shell commands; you should take every precaution possible to code your server so that the application does not cause any potential security problems. Many security problems on Linux come around as a result of some hacker finding a way to obtain superuser access. This avenue is usually due to applications that do not take the appropriate measures to make sure the application is secure.

Giving the Socket a Name

In the real world, it's a bit hard to listen for something when you don't know what that something is. Likewise, a server socket cannot listen for client connections unless the socket knows the address it is listening to. To accomplish this, the socket must be bound to a particular address using the `bind` function as shown in Listing 7.10. The first of the `bind` function parameters is the socket file descriptor that was returned after a successful call to the `socket` function. The second parameter is the address of the socket as a `TsockAddr` data type, which includes the IP addresses and port to listen to, and the third parameter is the size of the record type that was passed into the `__addr` parameter. You may not be able to bind to a particular socket if another process has already bound to that particular IP address port combination. As a result, only one server socket can bind to a particular address.

Listing 7.10 - The bind function

```

function bind(__fd: TSocket; const __addr: sockaddr; __len: socklen_t):Integer;
...

    //Bind the Socket

```

```

    if bind(ServerSocketID, ServerAddressToBindTo, sizeof(ServerAddressToBindTo)) = -1
    then
    begin
        perror('Could not bind to the address.');
```

```
        Exit;
```

```
    end;
```

Binding to a socket is often referred to as giving the socket a name, and defines what address and port a server socket will listen to. However, the bind function does not handle the serving of requests. To listen and accept connections, you need to work with the listen and accept functions.

Listening to and Accepting Connections

So far, to create your socket server, you have created the socket, defined the address, and given the socket a name. All that is needed now is to accept the connections and deal with connections as they arrive. Doing this involves listening for client socket connections on your port and then accepting each request that comes through.

Listing 7.11 - The listen function

```

function listen(__fd: TSocket; __n: Cardinal): Integer; cdecl;

    if listen(ServerSocketID, 10) = -1 then
    begin
        perror('Could not listen on the socket.');
```

```
        Exit;
```

```
    end;
```

The listen function states that a socket is now listening for client socket requests to the addresses that were passed to the bind function. The first parameter to the listen function is the socket file descriptor that was returned from the socket function and the second parameter is the number of connections that can be queued up or backlogged until error messages are returned to the clients who are attempting connections.

Now that the socket is listening, all that is left is to deal with the incoming client socket requests. This is done by the accept function shown in Listing 7.12.

Listing 7.12 - The accept function

```

function accept(__fd: TSocket; __addr: PSockAddr; __addr_len: PSocketLength): Integer;
```

When you call the accept function, the function will not return until there is a client socket that is requesting a connection to your socket service. When the accept function call returns, you will have the details of the client connection such as the address information and the file descriptor of the client socket.

The accept function takes three parameters: the __fd parameter, which contains the file descriptor of the server socket; __addr, which is a pointer to a TSockAddr record structure that will contain the address of the client that is attempting a connection to your server; and the __addr_len function, which is the size of the structure in the __addr parameter.

The accept function will return the file descriptor of the client socket, so you can read and write from the socket function in the same way that you would a file opened with the LibC.open function.

Listing 7.13 demonstrates the complete socket server that creates the socket, defines which address it listens to, and listens, accepts, and reads and writes to the client connection. This demonstrates accepting a client's request and then writing to that client the current date and time.

Listing 7.13 - The complete code for the DayTime server

```
program SimpleDateTimeServer;

(*
    This program is a simple time server that handles requests
    and returns the date and time back to the server.

    This example is for the "Tomes of Kylix - The Linux API"
    by Glenn Stephens published by Wordware (www.wordware.com)
*)

{$APPTYPE CONSOLE}

uses
    SysUtils, Classes, Types, Libc;

var
    //Server Variables
    ServerSocketID: integer;
    ServerAddressToBindTo: TSockAddrIn;

    //Client Socket Variables
    NewClientConnection: integer;
    SizeOfClientAddress: integer;

    //Variables for sending data back to the client
    ResponseToSend: string[35];
    ResponseData: PChar = nil;

begin
    writeln('Kicking off the DateTime server...');
    //Create the Socket
    ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);
    try
        //Define the Address to Connect to
        with ServerAddressToBindTo do
            begin
                sin_family := AF_INET;
                //listen to any IP address
                sin_addr.S_addr := htonl(INADDR_ANY);
                //listen on port 13
                sin_port := htons(13);
            end;

        //Bind the Socket
        if bind(ServerSocketID, ServerAddressToBindTo,
            sizeof(ServerAddressToBindTo)) = -1 then
            begin
                perror('Could not bind the address.');
```

```
Exit;
```

```

end;

if listen(ServerSocketID, 10) = -1 then
begin
    perror('Could not listen on the socket.');
```

Exit;

```
end;

writeln('DateTIme server is now listening for requests...');
```

SizeOfClientAddress := sizeof(TSockAddrIn);

```
writeln('DateTIme Server Listing.....');
```

repeat

```
    //This will wait until a connection is received
    NewClientConnection := accept(ServerSocketID,
        @NewClientConnection, @SizeOfClientAddress);
    try
        //Now we have a connection to the client, so we can send the DayTIme
        //back to the client and then close the socket.
```

 //Send the Response back to the Web server

```
    ResponseToSend := FormatDateTIme('ddd mmm dd hh:nn:ss yyyy', Now);
    ResponseData := StrAlloc(Length(ResponseToSend)+1);
    StrPCopy(ResponseData, ResponseToSend);

    //Send the data back to the client
    send(NewClientConnection, ResponseData^, StrLen(ResponseData)+1, 0);

finally
    StrDispose(ResponseData);
    _close(NewClientConnection);
end;
until false;

finally
    _close(ServerSocketID);
    writeln('Closed down the DayTIme server.');
```

end;

```
end.
```

Ways to Process a Client Socket

The code in Listing 7.13 demonstrates the aspects of creating a simple socket server that accepts connections and then sends the appropriate response. Although the server in Listing 7.13 demonstrates sending responses, it can only deal with one connection at a time and must send the response to the client before it can handle other clients.

For a simple server such as the DayTime server, handling individual requests may not seem so daunting, but imagine a server that returns data driven HTML content. A Web server may take 10 seconds to handle a single request. If 10 clients suddenly connect to a Web server, and all connections are handled one at a time, then the tenth connection would get serviced 100 seconds later, which would be unacceptable for a Web request.

For this Web server, you can use the fork function to separate the listening and processing sections of a Web server to make a more efficient socket server. The socket server is

created in much the same manner as was done with the DayTime server. However, when a client socket is accepted, the server process forks and the new process that is created handles the client socket, sends the information back to the client, and closes the client connection. The original process from the fork call simply loops and handles more client sockets so that it can fork another process to handle the client socket separately. This cycle of passing off client requests to forked processes means that the individual requests can be handled much more effectively within your application.

Listing 7.14 demonstrates a small Web server that handles HTML and text files that use this technique of forking. It uses the fork mechanism for handling client requests and you will see that the code is only slightly different from the DayTime server where it handles the request and that the passing off of the data processing is done in the child process.

Listing 7.14 - A simple HTTP socket server that handles client requests by forking

```
program SimpleWebServer;

(*
    This program is a simple Web application that
    accepts any Web request and returns the contents
    of a file only if it is .html, .htm, or .txt

    It demonstrates the use of forking within an
    application to allow another process to handle a
    new client connection.
*)

{$APPTYPE CONSOLE}

uses
    SysUtils, Classes, Types, Libc;

var
    //Server Variables
    ServerSocketID: integer;
    ServerAddressToBindTo: TSockAddrIn;

    //Client Socket Variables
    NewClientConnection: integer;
    SizeOfClientAddress: integer;

    //Misc
    ForkResult: integer = 0;
    MyBuffer: array[0..10000] of char;
    ContentLength: integer;
    ResponseToSend: string;
    WebFileDirectory: string;
    ResponseData: PChar;

function CalculateWebResponse(strWebRequest: string): string;
var
    Info: TStringList;
    DataToReturn: TStringList;
    counter: integer;
    strFilename: string;
```



```

    strErrorMessage: string;
    iPos: integer;
    strFileExt: string;
begin
    Result := '';

    DataToReturn := TStringList.Create;
    try
        Info := TStringList.Create;
        try
            Info.Text := strWebRequest;

            //There should be a line that is of the format GET /filename.htm HTTP/1.0
            //That is what we look for; otherwise we will return an error to the client
            strFilename := '';
            for counter := 0 to Info.Count - 1 do
                begin
                    iPos := Pos('GET ', Info[counter]);
                    if iPos <> 0 then
                        begin
                            //Here we get the file that we want to return.
                            strFilename := Copy(Info[counter], iPos + 4, Length(Info[counter]));
                            //The string will now look like "/filename.htm HTTP/1.0"
                            //so we only have to remove the HTTP to get the filename
                            iPos := Pos('HTTP', strFilename);
                            if iPos <> 0 then
                                strFilename := Trim(Copy(strFilename, 1, iPos-1))
                            else begin
                                strFilename := '';
                                strErrorMessage := 'The file could not be found.';
                            end;

                            break;
                        end;
                    end;
                end;

            //Add support for the index.html as default
            if strFilename = '/' then
                strFilename := '/index.html';

            //Let's see if the file exists first
            if not FileExists(WebFileDirectory+strFilename) then
                begin
                    //the file does not exist
                    strFilename := '';
                    strErrorMessage := 'The file could not be found.';
                end else begin
                    strFileExt := lowercase(ExtractFileExt(strFilename));
                    if not ((strFileExt = '.htm') or (strFileExt = '.html') or
                        (strFileExt = '.txt')) then
                        begin
                            //The file extension is invalid. Return an error
                            strErrorMessage := 'Invalid File Extension Type.';
                            strFilename := '';
                        end;
                    end;
                end;
            end;
        end;
    end;
end;

```

```

    if strFilename <> '' then
    begin
        //We can return the file that was requested back to the user
        writeln('Returning the web file: ', strFilename);
        DataToReturn.LoadFromFile(WebFileDirectory+strFilename);
        ContentLength := Length(DataToReturn.Text);
        DataToReturn.Insert(0, 'HTTP/1.0 200 OK');
        DataToReturn.Insert(1, '');
    end else begin
        //We will return the File not Found error message
        writeln('A Request was made for a file that does not exist.');
```

DataToReturn.Add('HTTP/1.0 404 Not Found');

DataToReturn.Add('');

DataToReturn.Add('<html><body><h1>HTTP/1.0 404 Not Found</h1><p>');

DataToReturn.Add('The file you requested is not located on this Web server ');

DataToReturn.Add('or is not an html or text file. This simple Web server for ');

DataToReturn.Add('the "Tomes of Kylix - Linux API" only deals with HTML and ');

DataToReturn.Add('text files only.<p>');

DataToReturn.Add('Error: '+strErrorMessage+'</body></html>');

end;

finally

Info.Free;

end;

DataToReturn.Add('');

DataToReturn.Add('');

Result := DataToReturn.Text;

finally

DataToReturn.Free;

end;

end;

begin

//Because we will be forking this application, we will ignore

//any child processes closing

signal(SIGCHLD, TSignalHandler(SIG_IGN));

//Set up the Web directory if it is not set up already

WebFileDirectory := GetCurrentDir;

if Copy(WebFileDirectory, Length(WebFileDirectory), 1) <> '/' then

WebFileDirectory := WebFileDirectory + '/';

WebFileDirectory := WebFileDirectory + 'webfiles/';

ForceDirectories(WebFileDirectory);

//Create the socket

ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);

try

//Define the Address to Connect to

with ServerAddressToBindTo do

begin

sa_family := AF_INET;

sin_addr.S_addr := htonl(INADDR_ANY);

sin_port := htons(91);

end;

//Bind the socket

```

    if bind(ServerSocketID, ServerAddressToBindTo, sizeof(ServerAddressToBindTo)) = -1
then
    begin
        perror('Could not bind the address.');
```

Exit;

```
    end;
```

//Accept connections to the socket

```
    if listen(ServerSocketID, 10) = -1 then
begin
    perror('Could not listen on the socket.');
```

Exit;

```
end;
```

//Get any new connection, fork, process the client in the forked child
//and then close the forked process

```
writeln('Server Listing.....');
```

repeat

```
    SizeOfClientAddress := sizeof(TSockAddrIn);
    NewClientConnection := accept(ServerSocketID, @NewClientConnection,
        @SizeOfClientAddress);
```

//When we get here, then we have a new socket connection, so we
//fork and handle the processing
//in the forked process

```
writeln('Received a new Web Request. Spawning a new child ->');
```

ForkResult := fork;

```
if ForkResult = -1 then
begin
    writeln('There was a problem forking a new process. Exiting.');
```

Exit;

```
end else if ForkResult = 0 then begin
    //We are the newly created process. Handle it and close.
    recv(NewClientConnection, MyBuffer, 10000, 0);
    ResponseToSend := CalculateWebResponse(MyBuffer);
```

//Send the response back to the Web server

```
    ResponseData := StrAlloc(Length(ResponseToSend)+1);
    StrPCopy(ResponseData, ResponseToSend);
    send(NewClientConnection, ResponseData^, StrLen(ResponseData)+1, 0);
    __close(NewClientConnection);
```

Exit;

```
end else begin
    //The Parent has no need for the client connection
    __close(NewClientConnection);
end;
```

until false;

```
finally
    __close(ServerSocketID);
    if ForkResult = 0 then
        writeln('Closed down a spawned process.')
```

else

```
writeln('Closed main listening process.');
```

```
end;
```

```
end.
```

The code in Listing 7.14 could very well be adapted to use threads instead of forked processes to improve efficiency. A pool of individual threads could be created when the process starts and then suspended until a new connection is ready. When a new connection occurs, the thread is resumed and deals with an individual client socket.

Using threads in this manner is more effective because creating a process takes longer than reestablishing an existing thread.

The DayTime server and the simple HTTP server demonstrate what is known as a stateless server, where the connection is made to the server, the information is retrieved, and then the connection is closed. Regrettably, not all servers will perform in this way. A server that tells which users are online would be one case where the client connections need to remain open. Whenever a user logs on, each connection needs to be notified that the user has logged on, so the client connection cannot be closed immediately.

One way this scenario could be handled is by using the select function to determine when the status of a file descriptor has changed. Because all sockets are file descriptors, we can use the select function to alert the process that either the server socket or one of the client sockets has changed status, which usually means that data has been read or written to them. For a demonstration of using the select function to wait on sockets, see Listing 7.33.

Look, Up in the Sky, It's the Internet Socket Server

One of the great things I love about Linux is its great built-in support for the Internet and the simplicity of building Internet services. The Internet daemon and extended Internet server that come with Linux are just such tools.

Up until now, you have been creating socket server applications that do much the same things. These socket servers create the socket, define the address to listen to and bind to it, listen and accept connections, and then deal with the client that is connected.

The only thing that is different for all servers is that the client socket will retrieve different data depending on what Internet services are requested. The Internet daemon or extended Internet daemon are examples of what is known as superservers. These superservers listen on particular addresses for client connections, and when the connections are obtained, the superservers fork and pass the processing of the client socket to a specific application designed to deal with the particular Internet service that was requested.

What the Internet daemon or extended Internet daemon does is handle the processing of incoming requests for clients, accept the connections, and then pass the processing of the client to your application. These daemons make effective use of the resources on the system by having one server that handles all client connections rather than having a process for every service offered on the machine.

The magic thing about the Internet daemon is that not only does it handle the processing of requests, but when it passes off the handling of clients to your process, it can actually redirect the connecting client socket's input and output to the new process's standard input and output. This means that you can write an Internet server using simple Pascal commands such as `readln` and `writeln`. Consider the case of the DayTime server that you wrote earlier. To implement the same server using the `inet` daemon or the extended Internet

daemon, all you would have to do is write a single line of text, which is the DayTime, out to the machine, shown in Listing 7.15.

Listing 7.15 - The datetime server written for use by the Internet daemon

```
program MyDateTimeServer;

{$APPTYPE CONSOLE}

uses SysUtils;

begin
    writeln(FormatDateTime('ddd mmm dd hh:nn:ss yyyy', Now));
end.
```

This is also true for any other Internet services. The simple HTTP server that we wrote earlier could be easily written as a number of readln and writeln function calls that make up the communication conversation, rather than writing to specific sockets.

Once the application is written, you only have to configure the application to make it work with the superserver. This requires that you modify the configuration files of either the Internet daemon or the extended Internet daemon. The Internet daemon's configuration file is normally located at /etc/inetd.conf and the extended Internet daemon's configuration file is located at /etc/xinetd.conf. There are major differences between the configuration files of the Internet daemon and the extended Internet daemon. The Internet daemon uses a single line in the configuration file to represent the application running shown in Listing 7.16. The extended Internet daemon, however, uses a block of text defining how the Internet service is to be implemented. Listing 7.17 demonstrates a simple implementation of the extended Internet daemon, although there are many more options available when using the extended Internet daemon.

Listing 7.16 - inetd.conf configuration for the Internet daemon

```
ftp stream tcp nowait root /libexec/ftpd ftpd
DayTime stream tcp nowait root /home/glenn/BookApps/MyDateTimeServer MyDateTimeServer
```

The entries in the inetd.conf file are the service, the style of the socket, the protocol to use, whether or not the process should wait, the username the process runs under, the program file, and any arguments to the application. The service is one of the Internet services listed in the /etc/services configuration file and the service must be in this file or the service will not work. The style will be either tcp for a Transmission Control Protocol connection or udp for a User Datagram Protocol connection. Using wait for the value specifies if the process should fork for every new connection, whereas using nowait means that the process that is created will actually handle the socket connection process. Using the nowait option means, however, that the process spawned by the Internet daemon will have to handle all connections using sockets and the connecting socket will not be redirected to the new process's standard input and output.

The configuration file for the extended Internet daemon is really the same information as the Internet daemon only formatted in a different way. The extended Internet daemon, however, allows much more flexibility in areas such as logging and security. For most configurations, you will use similar data to what is contained in the inetd.conf file. A handy way to convert an Internet daemon configuration file to an extended Internet configuration

file is by using one of the utilities that ship with the extended Internet daemon. Depending on the version of xinetd that is installed on your Linux distribution there are normally three options to convert the file. There may be an application called inetdconvert or itox that will convert the inetd.conf for use with xinetd. You may also have a Perl script named xconv.pl which also performs the conversion of the configuration file. Regardless of the utility that you use, you will find that converting the process is a snap.

Listing 7.17 - xinetd.conf configuration changes for the extended Internet daemon

```
#
# Simple configuration file for xinetd
#

defaults
{
    instances            = 60
    log_type             = SYSLOG authpriv
    log_on_success        = HOST PID
    log_on_failure        = HOST RECORD
}

service DayTime
{
    socket_type = stream
    wait        = no
    user        = root
    server       = /home/glenn/BookApps/MyDateTimeServer
    server_args =
}
```

Integrating with the Visual Socket Components

You may at some time have a need to mix raw socket API functionality with some of the CLX components in Kylix. Although components such as TClientSocket and TServerSocket handle much of the work for processing sockets, there may be a time when a socket file descriptor is needed to be passed out to a third-party library that will process the socket request.

If you have an open socket, regardless of whether it is a client or server socket, you can use the Handle property of the TClientSocket or TServerSocket to obtain the file descriptor. You can then use the file descriptor to read and write data using the techniques shown in this chapter. When you are running a server using the TServerSocket, the OnAccept event contains the ClientSocket. With this object you can then use ClientSocket.Handle to obtain the socket file descriptor of the socket, which you can pass to most of the Linux API socket functions.

API Reference

accept *LibC.pas*

Syntax

```
function accept(
  __fd: TSocket;
  __addr: PSockAddr;
  __addr_len: PSocketLength
):Integer;
```

Description

After a server socket has been created, has been bound to an address, and is listening for a connection, the `accept` function takes the first client connection from the queue of pending connections and returns a valid file descriptor that represents the connecting client socket.

Parameters

`__fd`: This is the file descriptor of the server socket that was created with the `socket` function.

`__addr`: The `__addr` parameter is a pointer to a buffer that after the function call will hold the address of the client socket.

`__addr_len`: The `__addr_len` function is a pointer to a cardinal value that will hold the size of the address structure returned in the `__addr` parameter.

Return Value

If the function is successful, it returns a valid descriptor that represents the newly connected client socket. If the function fails, it returns `-1`.

See Also

`socket`, `bind`, `listen`

Example

See Listing 7.13 - The complete code for the DayTime server.

bind *LibC.pas*

Syntax

```
function bind(
  __fd: TSocket;
  const __addr: sockaddr;
  __len: socklen_t
):Integer;
```

Description

The bind function assigns a specific address to a socket. Using the bind function to assign an address is also known as “naming a socket.” This function is most often used to define the addresses that a socket will listen on when the socket is used as a server socket. Naming a socket is the step right before a socket server starts listening and accepting client connections.

Parameters

`__fd`: This is the file descriptor of the socket to be used as a server socket. This parameter is normally the result of a successful call to the socket function.

`__addr`: This is the address information that the socket will use. Any client socket connecting to an address matching this parameter will connect to the server socket in the `__fd` parameter.

`__len`: This is the size of the address structure in the `__addr` parameter.

Return Value

When this function is successful, it returns 0. If the function is unsuccessful, which is normally a result of another socket already being bound to that address or the file descriptor being invalid or not a socket file descriptor, the function returns -1.

See Also

accept, listen, socket

Example

Listing 7.18 - Using the bind function

```
var
  ServerSocketID: integer;
  ServerAddressToBindTo: TSockAddrIn;
begin
  //Create the Socket
  ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);

  //Define the Address to Connect to
  with ServerAddressToBindTo do
  begin
    sin_family := AF_INET;
    //listen to any IP address
    sin_addr.S_addr := htonl(INADDR_ANY);
    sin_port := htons(80);
  end;

  //Bind the Socket
  if bind(ServerSocketID, ServerAddressToBindTo,
    sizeof(ServerAddressToBindTo)) = -1 then
  begin
    perror('Could not bind the address.');
```



```

        Exit;
    end;

    //...

end;

```

connect *LibC.pas*

Syntax

```

function connect(
    __fd: TSocket;
    const __addr: sockaddr;
    __len: socklen_t
):Integer;

```

Description

The connect function attaches a valid socket file descriptor to a connecting server, where the server is defined to be at the address defined in the `__addr` parameter. This function is normally used to connect a client socket to a server. The connection of a socket is done before the socket can be used to read and write to the connected server. When the socket is using UDP as its transport, the connect function defines the default location from which UDP messages are sent and received.

Parameters

`__fd`: This is the file descriptor of a valid socket. This parameter is typically the result of a successful call to the socket function.

`__addr`: This is a TSockAddr structure that defines the address of the server socket to connect to.

`__len`: This parameter defines the size of the TSockAddr structure used in the `__addr` parameter.

Return Value

When this function is successful, it returns 0. If the function fails, it returns `-1`. A failure for this function is typically caused by the file descriptor not being a valid file descriptor or socket, the address being invalid, or a server not accepting connections.

See Also

socket, `__close`, send, recv

Example

See Listing 7.6 - Code to connect to a time server.

endhostent LibC.pas**Syntax**

procedure endhostent;

Description

The endhostent function closes a connection to the host's database that was previously opened with the sethostent function.

See Also

gethostent, sethostent

Example

See Listing 7.21 - Using the gethostent function.

endservent LibC.pas**Syntax**

procedure endservent;

Description

The endservent function closes the /etc/services file that was previously opened by a call to setservent or getservent, so entries could be received from the /etc/services file as a TServEnt record structure. This function can be likened to the __close function, but it will close the /etc/services file that was previously opened.

Example

See Listing 7.27 - Using the getservent function.

gethostbyaddr LibC.pas**Syntax**

```
function gethostbyaddr(  
  __addr: Pointer;  
  __len: __socklen_t;  
  __type: Integer  
):PHostEnt;
```

Description

The gethostbyaddr function returns the location of the THostEnt that matches the address in the host's file, /etc/hosts, or from a name database such as a Domain Name System (DNS). The function passes in the address of the host as either a string or an address data type.

Parameters

__addr: This is a pointer value either to a null-terminated string that contains the address in readable form, such as the IPv4 address 205.21.43.2, or a pointer to a TSocketAddr record.

`__len`: This is the size in bytes of the address in the `__addr` parameter.

`__type`: This is the socket family type that is used such as `AF_INET` for an IPv4 address or `AF_INET6` for an IPv6 address.

Return Value

When the host entry can be found, the function will return a pointer to a `THostEnt` record which contains information such as the name of the host, the aliases used for the host, the type of address, and a list of addresses for the host from the server used. The `THostEnt` record is defined as follows:

```
type
    hostent = record
        h_name: PChar;
        h_aliases: PPChar;
        h_addrtype: Integer;
        h_length: socklen_t;
        case Byte of
            0: (h_addr_list: PPChar);
            1: (h_addr: PPChar);
        end;
    THostEnt = hostent;
    PHostEnt = ^THostEnt;
```

The `h_name` field represents the name of the host. The `h_aliases` field represents a list of aliases used for the host. `h_addrtype` represents the address family, which is normally `AF_INET`. The `h_length` field represents the length of the address in bytes, and `h_addr_list` and `h_addr` represent the address list of the hosts.

If the host entry cannot be found, the function will return `nil` and the reason that the entry could not be found will be found by calling the `h_errno` function, which will return one of the following host error values:

`NETDB_INTERNAL`: An internal error occurred during the search.

`HOST_NOT_FOUND`: The host could not be found.

`TRY_AGAIN`: Try again.

`NO_RECOVERY`: A non-recoverable error happened during the search.

See Also

`gethostbyname`, `gethostbyname2`, `gethostent`

Example

Listing 7.19 - Using the `gethostbyaddr` function

```
procedure TfrmHostIterator.btnFindByAddressClick(Sender: TObject);
var
    pAddress: array[0..100] of char;
    Host: PHostEnt;
    Aliases: TStringList;
    AddressFamily: integer;
begin
    //Find the host based on the host name
    StrPCopy(pAddress, edHostName.Text);
```

```

if cbAddressFamily.ItemIndex = 0 then
  AddressFamily := AF_INET
else
  AddressFamily := AF_INET6;

//The first parameter of gethostbyname is a pointer so it can take either a
//PChar data type or a pointer to a TSocketAddr
Host := gethostbyaddr(@pAddress, StrLen(pAddress), AddressFamily);

if Host = nil then
begin
  mmSearchResults.Lines.Clear;
  mmSearchResults.Lines.Add('Could not find the host.');
```

mmSearchResults.Lines.Add('Reason:');

```

  case h_errno of
    NETDB_INTERNAL: mmSearchResults.Lines.Add('Internal Error');
    HOST_NOT_FOUND: mmSearchResults.Lines.Add('The host could not be found');
    TRY_AGAIN: mmSearchResults.Lines.Add('Try again.');
```

NO_RECOVERY: mmSearchResults.Lines.Add('Non recoverable errors');

```

  end;
end else begin
  Aliases := TStringList.Create;
  try
    mmSearchResults.Lines.Clear;
    mmSearchResults.Lines.Add('Name: ' + StrPas(Host^.h_name));
    PPCharToStringList(Host^.h_aliases, Aliases);
    mmSearchResults.Lines.Add('Aliases: ' + Aliases.CommaText);
  finally
    Aliases.Free;
  end;
end;
end;
```

gethostbyname LibC.pas

Syntax

```

function gethostbyname(
  __name: PChar
): PHostEnt;
```

Description

The `gethostbyname` function will return a pointer to a `THostEnt` structure which contains information on a network host that is stored in the `/etc/hosts` file.

Parameters

`__name`: This is the name of the host to look up. This can be either the name of the host or the IP address of the host.

Return Value

When successful, the function will return a pointer to a `THostEnt` record, which contains information such as the name of the host, the aliases used for the host, the type of address,

and a list of addresses for the host from the server used. The THostEnt record is defined as follows:

```
type
  hostent = record
    h_name: PChar;
    h_aliases: PPChar;
    h_addrtype: Integer;
    h_length: socklen_t;
    case Byte of
      0: (h_addr_list: PPChar);
      1: (h_addr: PPChar);
    end;
  THostEnt = hostent;
  PHostEnt = ^THostEnt;
```

The `h_name` field represents the name of the host. The `h_aliases` field represents a list of aliases used for the host. `h_addrtype` represents the address family, which is normally `AF_INET`. The `h_length` field represents the length of the address in bytes, and `h_addr_list` and `h_addr` represent the address list of the hosts.

If the host could not be found, the function will return a nil pointer.

See Also

`gethostbyname2`

Example

Listing 7.20 - Using the `gethostbyname` function

```
procedure TfrmHostIterator.btnFindHostByNameClick(Sender: TObject);
var
  pHostName: array[0..100] of char;
  Host: PHostEnt;
  Aliases: TStringList;
  AddressFamily: integer;
begin
  //Find the host based on the host name
  StrPCopy(pHostName, edHostName.Text);
  if not cbSpecifyAddressFamily.Checked then
    Host := gethostbyname(pHostName)
  else begin
    if cbAddressFamily.ItemIndex = 0 then
      AddressFamily := AF_INET
    else
      AddressFamily := AF_INET6;
    Host := gethostbyname2(pHostName, AddressFamily);
  end;

  if Host = nil then
    mmSearchResults.Lines.Text := 'Could not find the host.'
  else begin
    Aliases := TStringList.Create;
    try
      mmSearchResults.Lines.Clear;
      mmSearchResults.Lines.Add('Name: ' + StrPas(Host^.h_name));
      PPCharToStringList(Host^.h_aliases, Aliases);
```

```

        mmSearchResults.Lines.Add('Aliases: ' + Aliases.CommaText);
    finally
        Aliases.Free;
    end;
end;
end;
end;

```

gethostbyname2 LibC.pas

Syntax

```

function gethostbyname2(
    __name: PChar;
    __af: Integer
): PHostEnt;

```

Description

The `gethostbyname2` function is similar to the `gethostbyname` function, but this function will specify the address family to use, such as `AF_INET` for IPv4 or `AF_INET6` for IPv6.

Parameters

`__name`: This parameter is the name of the host to look up. This can be either the name of the host or the IPv4 or IPv6 address of the host.

`__af`: This is the address family that is used for the host lookup. This value will be `AF_INET` for IPv4 or `AF_INET6` when IPv6 is used as the address family.

Return Value

When successful, the function will return a pointer to a `THostEnt` record, which contains information such as the name of the host, the aliases used for the host, the type of address, and a list of addresses for the host from the server used. The `THostEnt` record is defined as follows:

```

type
    hostent = record
        h_name: PChar;
        h_aliases: PPChar;
        h_addrtype: Integer;
        h_length: socklen_t;
        case Byte of
            0: (h_addr_list: PPChar);
            1: (h_addr: PPChar);
        end;
    THostEnt = hostent;
    PHostEnt = ^THostEnt;

```

The `h_name` field represents the name of the host. The `h_aliases` field represents a list of aliases used for the host. `h_addrtype` represents the address family, which is normally `AF_INET`. The `h_length` field represents the length of the address in bytes, and `h_addr_list` and `h_addr` represent the address list of the hosts.

If the host could not be found, the function will return a nil pointer.

See Also

gethostbyname, gethostent

Example

See Listing 7.20 - Using the gethostbyname function.

gethostent LibC.pas*Syntax*

```
function gethostent:PHostEnt;
```

Description

gethostent will retrieve a pointer to a THostEnt from the /etc/hosts file, opening the file if the file has not been opened already by a call to gethostent or sethostent. Successive calls to this function will return the next host entry in the /etc/hosts file until no more entries are found, in which case the function will return a nil pointer. Using this function is similar to reading from a file; however, this function will be reading from the /etc/hosts file specifically.

Return Value

When there are still more host entries in the /etc/hosts file, the function will return a pointer to a THostEnt record, which contains information such as the name of the host, the aliases used for the host, the type of address, and a list of addresses for the host from the server used. The THostEnt record is defined as follows:

```
type
  hostent = record
    h_name: PChar;
    h_aliases: PPChar;
    h_addrtype: Integer;
    h_length: socklen_t;
    case Byte of
      0: (h_addr_list: PPChar);
      1: (h_addr: PPChar);
    end;
  THostEnt = hostent;
  PHostEnt = ^THostEnt;
```

The h_name field represents the name of the host. The h_aliases field represents a list of aliases used for the host. h_addrtype represents the address family, which is normally AF_INET. The h_length field represents the length of the address in bytes, and h_addr_list and h_addr represent the address list of the hosts.

If there are no more hosts left in the file to read, the function will return nil.

See Also

sethostent

*Example***Listing 7.21 - Using the gethostent function**

```

procedure TfrmHostIterator.DisplayHosts;
var
  Host: PHostEnt;
  HostItem: TListItem;
  Aliases: TStringList;
begin
  //This displays the current hosts on the system
  lvHosts.Items.Clear;

  Aliases := TStringList.Create;
  try
    //Iterate through all the Hosts
    sethostent(0);
    Host := gethostent;
    while Host <> nil do
    begin
      HostItem := lvHosts.Items.Add;
      HostItem.Caption := StrPas(Host^.h_name);
      PPCharToStringList(Host^.h_aliases, Aliases);
      HostItem.SubItems.Add(Aliases.CommaText);
      Host := gethostent;
    end;
  endhostent;
finally
  Aliases.Free;
end;
end;

```

gethostname LibC.pas***Syntax***

```

function gethostname(
  Name: PChar;
  Len: socklen_t
): Integer;

```

Description

The gethostname function returns the hostname of the current system.

Parameters

Name: This is a null-terminated string that points to a buffer that will hold the name of the host.

Len: After a successful call, this function will hold the length of the string returned in the Name parameter.

Return Value

If the function successfully returns the hostname, the function will store the hostname in the Name parameter and the function will return 0. If the function fails, it returns -1.

See Also

gethostent, gethostbyname, gethostbyname2

Example

Listing 7.22 - Using the gethostname function

```
var
  pThisHost: array[0..100] of char;
begin
  if gethostname(pThisHost, 100) <> -1 then
    lblCurrentHostName.Caption := pThisHost;
end;
```

getpeername LibC.pas

Syntax

```
function getpeername(
  __fd: TSocket;
  var __addr: sockaddr;
  var __len: socklen_t
):Integer;
```

Description

The getpeername function will return the address of a connected socket passed into the __fd parameter. Normally, you will use this function to determine the address of a server the socket is connected to when the only information you have is the socket file descriptor.

Parameters

__fd: This is the file descriptor of the socket.

__addr: This is a TSocketAddr record which after a successful function call will contain the address information of the socket.

__len: This parameter is an integer variable that should initially contain the size in bytes of the address passed into the __addr parameter. After the function call returns, this parameter will contain the actual size of the address that was returned in the __addr parameter.

Return Value

When the function is successful, it returns 0. If the function fails for any reason, it will return -1. The function will normally only fail if the file descriptor is not a valid file descriptor, not a valid socket, or the socket is not connected.

See Also

getsockname

*Example***Listing 7.23 - Using the getpeername function**

```

var
  NewAddr: TSocketAddr;
  NewAddrSize: Cardinal;

begin
  //...,
  //Let's look at who we are connected to
  getpeername(SocketFileDescriptor, NewAddr, NewAddrSize);
  writeln('Connection from ', inet_ntoa(NewAddr.sin_addr), ':',
    ntohs(NewAddr.sin_port));
end;

```

getprotobyname LibC.pas*Syntax*

```

function getprotobyname(
  __name: PChar
): PProtoEnt;

```

Description

This function returns information on a protocol found in the `/etc/protocols` file when only the name of the protocol is known. All information about the protocol is returned, including the name of the protocol, the aliases used for the protocol name, and the protocol number.

Parameters

`__name`: This is the name of the protocol in the `/etc/services` file for which to search.

Return Value

If the protocol can be found, the function returns a pointer to a `TProtoEnt` record structure. The `TProtoEnt` record is defined as follows:

```

type
  protoent = record
    p_name: PChar;
    p_aliases: ^PChar;
    p_proto: u_short;
  end;
  TProtoEnt = protoent;
  PProtoEnt = ^TProtoEnt;

```

Within the `TProtoEnt` record, the `p_name` field refers to the name of the protocol, the `p_aliases` field refers to the list of aliases that can be used for the protocol, and the `p_proto` field represents the protocol number.

If the protocol cannot be found, the function returns `nil`.

See Also

`getprotobyname`

*Example***Listing 7.24 - Using the `getprotobyname` function**

```

uses
  Libc, LibcHelperFunctions;

procedure TfrmFindProtocol.btnFindClick(Sender: TObject);
var
  Protocol: PProtoEnt;
  Aliases: TStringList;
  ProtoName: array[0..100] of char;
begin
  if rbProtocolName.Checked then
  begin
    StrPCopy(ProtoName, edProtoName.Text);
    Protocol := getprotobyname(ProtoName);
  end else
    Protocol := getprotobynumber(strtoint(edProtoNo.Text));

  if Protocol <> nil then
  begin
    mmDetails.Lines.Add('Protocol Name: '+Protocol^.p_name);
    mmDetails.Lines.Add('Protocol Number: '+inttostr(Protocol^.p_proto));
    Aliases := TStringList.Create;
    try
      PPCharToStringList(System.PPChar(Protocol^.p_aliases), Aliases);
      mmDetails.Lines.Add('Aliases: '+Aliases.CommaText);
    finally
      Aliases.Free;
    end;
  end else
    MessageDlg('The Protocol could not be found.', mtInformation, [mbOk], 0);
end;

```

`getprotobynumber` LibC.pas*Syntax*

```

function getprotobynumber(
  __proto: Integer
):PProtoEnt;

```

Description

This function locates a protocol in the `/etc/protocols` file when only the protocol number is known. All information about the protocol is returned including the name of the protocol, the aliases used for the protocol name, and the protocol number.

Parameters

`__proto`: This parameter is the protocol number on which to perform the search.

Return Value

If the protocol can be found, the function returns a pointer to a TProtoEnt record structure. The TProtoEnt record is defined as follows:

```
type
  protoent = record
    p_name: PChar;
    p_aliases: ^PChar;
    p_proto: u_short;
  end;

  TProtoEnt = protoent;
  PProtoEnt = ^TProtoEnt;
```

Within the TProtoEnt record, the p_name field refers to the name of the protocol, the p_aliases field refers to the list of aliases that can be used for the protocol, and the p_proto field represents the protocol number.

If the protocol cannot be found, the function returns nil.

See Also

getprotobyname

Example

See Listing 7.24 - Using the getprotobyname function.

getservbyname LibC.pas

Syntax

```
function getservbyname(
  __name: PChar;
  __proto: PChar;
):PServEnt;
```

Description

The getservbyname function returns the details from the /etc/services file when the only information known about the service is the name of the service and the protocol used for the service.

Parameters

__name: This parameter is a null-terminated string that holds the name of the service to find.

__proto: This parameter is a null-terminated string that contains the protocol that is used for the service.

Return Value

If the service can be found within the /etc/services file, the function returns a pointer to a TServEnt structure. If the service could not be located, the function returns a nil value. The TServEnt details are defined as follows:

```

type
  servent = record
    s_name: PChar;
    s_aliases: PPChar;
    s_port: Integer;
    s_proto: PChar;
  end;
  TServEnt = servent;
  PServEnt = ^TServEnt;

```

s_name represents the name of the service, s_aliases refers to alternative names that can be used to represent the service, s_port represents the network byte ordered port number, and s_proto represents a string of the protocol that uses the port.

See Also

setservent, endservent, getservbyport

Example

Listing 7.25 - Using the getservbyname function

```

uses
  Libc, LibcHelperFunctions;

procedure TfrmService.btnFindClick(Sender: TObject);
var
  Service: PServEnt;
  Aliases: TStringList;
  Proto, ServiceName: array[0..100] of char;
begin
  StrPCopy(Proto, cbProtocol.Text);
  StrPCopy(ServiceName, cbServiceName.Text);
  Service := getservbyname(ServiceName, Proto);
  if Service <> nil then
    begin
      mmDetails.Lines.Add('Service Name: ' + Service^.s_name);
      mmDetails.Lines.Add('Port: '+inttostr(ntohs(Service^.s_port)));
      mmDetails.Lines.Add('Protocol: ' + Service^.s_proto);
      Aliases := TStringList.Create;
      try
        PPCharToStringList(Service^.s_aliases, Aliases);
        mmDetails.Lines.Add('Aliases: '+Aliases.CommaText);
      finally
        Aliases.Free;
      end;
    end else
      MessageDlg('The Service could not be found.', mtInformation, [mbOk], 0);
end;

```

getservbyport *LibC.pas*

Syntax

```

function getservbyport(
  __port: Integer;
  __proto: PChar

```

```
);PServEnt;
```

Description

The `getservbyport` function returns the details of a server when only the port number and protocol of the service are known. The function scans the `/etc/services` file to locate the information about the service and returns the information about the structure such as the name of the service, the port number, the protocol used, and any alias that can be used for the service.

Parameters

`__port`: This is the port number that the service should be found on, passed as a network byte ordered value.

`__proto`: This is the protocol of the service which is typically either `tcp`, `udp`, or another protocol located in the `/etc/protocols` file.

Return Value

The function returns a pointer to a `TServEnt` structure if it can locate the service. If the service could not be located, the function returns a `nil` value. The `TServEnt` details are as follows:

```
type
  servent = record
    s_name: PChar;
    s_aliases: PPChar;
    s_port: Integer;
    s_proto: PChar;
  end;
  TServEnt = servent;
  PServEnt = ^TServEnt;
```

`s_name` represents the name of the service, `s_aliases` refers to alternative names that can be used to represent the service, `s_port` represents the network byte ordered port number, and `s_proto` represents a string of the protocol that uses the port.

See Also

`setservent`, `endservent`, `getservbyname`

Example

Listing 7.26 - Using the `getservbyport` function

```
uses
  Libc, LibcHelperFunctions;

procedure TfrmServiceFinder.btnDisplayServiceClick(Sender: TObject);
var
  Service: PServEnt;
  Aliases: TStringList;
  Proto: array[0..20] of char;
begin
  StrPCopy(Proto, cbProtocol.Text);
```

```

Service := getservbyport(htons(strtoint(edPortNo.Text)), Proto);
if Service <> nil then
begin
  mmDetails.Lines.Add('Service Name: ' + Service^.s_name);
  mmDetails.Lines.Add('Port: '+inttostr(ntohs(Service^.s_port)));
  mmDetails.Lines.Add('Protocol: ' + Service^.s_proto);
  Aliases := TStringList.Create;
  try
    PPCharToStringList(Service^.s_aliases, Aliases);
    mmDetails.Lines.Add('Aliases: '+Aliases.CommaText);
  finally
    Aliases.Free;
  end;
end else
  MessageDlg('The Service could not be found.', mtInformation, [mbOk], 0);
end;

```

getservent *LibC.pas*

Syntax

```
function getservent:PServEnt;
```

Description

The function returns the next line read in from the `/etc/services` file, which contains a list of the machine's services and details. This function is normally called after `setservent` to reset the reading position of the `/etc/services` file to the start of the file. The function returns a pointer to a `TServEnt` record, which contains the name, port, and protocol of the service along with the aliases for that service.

Return Value

The function returns a pointer to a `TServEnt` structure until it reaches the last entry in the list. When no more services are available, the function returns `nil`.

```

type
  servent = record
    s_name: PChar;
    s_aliases: PPChar;
    s_port: Integer;
    s_proto: PChar;
  end;
  TServEnt = servent;
  PServEnt = ^TServEnt;

```

`s_name` represents the name of the service, `s_aliases` refers to alternative names that can be used to represent the service, `s_port` represents the network byte ordered port number, and `s_proto` represents a string of the protocol that uses the port.

See Also

`setservent`, `endservent`, `getservbyport`, `getservbyname`

*Example***Listing 7.27 - Using the getservent function**

```

uses
  Libc, LibcHelperFunctions;

procedure TfrmServices.DisplayServices;
var
  Service: PServEnt;
  ServiceItem: TListItem;
  Aliases: TStringList;
begin
  lvServices.Items.Clear;

  Aliases := TStringList.Create;
  try
    //Iterate through all the Services
    setservent(0);
    Service := getservent;
    while Service <> nil do
      begin
        ServiceItem := lvServices.Items.Add;
        ServiceItem.Caption := StrPas(Service^.s_name);
        ServiceItem.SubItems.Add(inttostr(ntohs(Service^.s_port)));
        ServiceItem.SubItems.Add(Service^.s_proto);
        PPCharToStringList(Service^.s_aliases, Aliases);
        ServiceItem.SubItems.Add(Aliases.CommaText);
        Service := getservent;
      end;
    endservent;
  finally
    Aliases.Free;
  end;
end;

```

getsockname LibC.pas***Syntax***

```

function getsockname(
  __fd: TSocket;
  var __addr: sockaddr;
  var __len: socklen_t
):Integer;

```

Description

The getsockname function returns the name of the socket. A socket is named when it is bound to a specific address/port combination as a result of calling the bind function on the socket. Normally within an application, you would have set the name of the socket already. The getsockname function is usually used to obtain the address information when only the socket file descriptor is known.

Parameters

`__fd`: This parameter is the file descriptor of a socket that is already bound to an address.

`__addr`: This is a pointer to a `TSockAddr` structure that after a successful function call will contain the address the socket is bound to.

`__len`: This parameter is a `socklen_t` variable that defines the length in size of the structure in the `__addr` parameter. When the function returns, the variable will actually hold the size of the structure returned in the `__addr` function.

Return Value

When successful, this function will return 0 and the address information will be stored in the `__addr` parameter and the size of the structure in `__addr` will be returned in the `__len` parameter. If the function is unsuccessful, it returns -1 and the error code can be found by a call to the `errno` function.

See Also

`bind`, `getpeername`

Example

Listing 7.28 - Using the `getsockname` function

```
procedure DisplaySocketAddress(SocketFD: integer);
var
  addr: TSockAddr;
  addrSize: Cardinal;
begin
  //This function displays the name (Address/port) that a socket is
  //bound to. This function is written for IPv4 sockets only.
  addrSize := sizeof(TSockAddr);
  if getsockname(SocketFD, addr, addrSize) = 0 then
    begin
      //We have the name of the socket
      if addr.sin_family = AF_INET then
        begin
          writeln('The socket is bound to:');
          write(' IP Address: ');
          if addr.sin_addr.S_addr = htonl(INADDR_ANY) then
            writeln('All local IP addresses.')
          else
            with addr.sin_addr.S_un_b do
              writeln(Format('%d.%d.%d.%d', [s_b1,
                s_b2, s_b3, s_b4]));
            writeln(' Port: ', ntohs(addr.sin_port));
          end else
            writeln('This function only works for the IPv4 sockets.');
```

getsockopt LibC.pas**Syntax**

```
function getsockopt(
  __fd: TSocket;
  __level: Integer;
  __optname: Integer;
  __optval: Pointer;
  var __optlen: socklen_t
):Integer;
```

Description

The `getsockopt` function returns the current options on an existing socket. The function can return information such as the size of the send and receive buffers, the type of socket, socket lifetime, routing options, and other available options.

Parameters

`__fd`: This is the socket file descriptor used to retrieve the options.

`__level`: This is the level of the socket option, and represents the category of option, such as if you want to examine the socket options or TCP options. The Kylix implementation of LibC.pas only includes the socket option level `SOL_SOCKET`, which is used for examining socket options. Although there are other available values for this parameter, they are not declared in Kylix's implementation of the LibC unit.

`__optname`: This parameter is the option name that will be examined and is a value from the following list. Most of these options will return an integer value representing true for non-zero values and false for 0, or an integer value specifying the option.

`SO_REUSEADDR`: This option specifies whether or not a server can bind more than once to a particular port.

`SO_KEEPALIVE`: Using this option will result in the socket that it is connected to sending messages on a regular basis, ensuring that the connection is still active.

`SO_LINGER`: This option retrieves the outcome of what will happen to a reliable socket when closed if data is still waiting to be sent from that socket.

`SO_BROADCAST`: This option specifies if UDP broadcasts are possible from this socket.

`SO_OOBINLINE`: This specifies if out-of-band data can be read in by the application in the same way normal data is read, using the `__write` or `recv` functions.

`SO_SNDBUF`: This option will return the size of the send buffer of the socket.

`SO_RCVBUF`: This option will return the size of the receive buffer of the socket.

`SO_TYPE`: This option will return the type of the socket, which may be 1 for a TCP socket and 2 for a UDP socket.

`SO_ERROR`: Reading this option will clear the error status of the socket and return the error status of the socket before the call.

SO_DONTROUTE: Reading this option will specify whether or not the socket is allowed to route out of the local network.

__optval: This parameter is a pointer to whatever record structure is used to retrieve the information about the particular option requested in the __level/__optname parameter combination. For example, when an integer is used, this field will be a pointer to a valid integer variable.

__optlen: The __optlen parameter is the length of the data in bytes of the structure passed into the __optval parameter. Normally, for an integer, this value would be sizeof(integer).

Return Value

When this function is successful, it returns 0. If the function fails for any reason, which is normally due to an invalid value in the __level or __optname fields, it will return -1.

See Also

setsockopt

Example

Listing 7.29 - Using the getsockopt function

```
procedure TfrmSockOptions.DisplaySocketOptions(fdSocket: integer);
var
  rVal: integer;
  ValueRead: integer;
  LingerValue: linger;
  OptionalLength: Cardinal;
begin
  //Display the options for the socket

  //Get SO_REUSEADDR options
  OptionalLength := sizeof(integer);
  rVal := getsockopt(fdSocket, SOL_SOCKET, SO_REUSEADDR, @ValueRead, OptionalLength);
  lblReuseAddr.Caption := inttostr(ValueRead);

  //Get SO_KEEPALIVE
  rVal := getsockopt(fdSocket, SOL_SOCKET, SO_KEEPALIVE, @ValueRead, OptionalLength);
  lblKeepAlive.Caption := inttostr(ValueRead);

  //Get SO_LINGER
  OptionalLength := sizeof(linger);
  rVal := getsockopt(fdSocket, SOL_SOCKET, SO_LINGER, @LingerValue, OptionalLength);
  lblLinger.Caption := 'On Status: '+inttostr(LingerValue.l_onoff)+'; Timeout: '+
    inttostr(LingerValue.l_linger);

  //Get SO_BROADCAST
  OptionalLength := sizeof(integer);
  rVal := getsockopt(fdSocket, SOL_SOCKET, SO_BROADCAST, @ValueRead, OptionalLength);
  lblBroadcast.Caption := inttostr(ValueRead);

  //Get SO_OOBINLINE
  rVal := getsockopt(fdSocket, SOL_SOCKET, SO_OOBINLINE, @ValueRead, OptionalLength);
  lblOOBInLine.Caption := inttostr(ValueRead);
```

```

//Get SO_SNDBUF
rVal := getsockopt(fdSocket, SOL_SOCKET, SO_SNDBUF, @ValueRead, OptionalLength);
lblSndBuf.Caption := inttostr(ValueRead);

//Get SO_RCVBUF
rVal := getsockopt(fdSocket, SOL_SOCKET, SO_RCVBUF, @ValueRead, OptionalLength);
lblRcvBuf.Caption := inttostr(ValueRead);

//Get SO_TYPE
rVal := getsockopt(fdSocket, SOL_SOCKET, SO_TYPE, @ValueRead, OptionalLength);
lblType.Caption := inttostr(ValueRead);

//Get SO_ERROR
rVal := getsockopt(fdSocket, SOL_SOCKET, SO_ERROR, @ValueRead, OptionalLength);
lblError.Caption := inttostr(ValueRead);

//Get SO_DONTROUTE
rVal := getsockopt(fdSocket, SOL_SOCKET, SO_DONTROUTE, @ValueRead, OptionalLength);
lblDontRoute.Caption := inttostr(ValueRead);

end;

procedure TfrmSockOptions.btnGetSockOptionsTCPClick(Sender: TObject);
var
    MySocket: TSocket;
begin
    //Let's create a TCP socket
    MySocket := socket(AF_INET, SOCK_STREAM, 0);

    //Let's check its options
    DisplaySocketOptions(MySocket);

    //Close the socket
    _close(MySocket);
end;

```

htonl LibC.pas

Syntax

```

function htonl(
    __hostlong: uint32_t
):uint32_t;

```

Description

This function converts an unsigned 32-bit integer value which is valid on the host system to a network byte ordered value that is suitable for communicating 32-bit integers across a network. For more information on network byte ordering, see the section titled “Socket Addresses” in this chapter.

Parameters

__hostlong: This is the integer value on the host system, which needs to be replaced with a network byte ordered value.

Return Value

The function returns the network byte ordered value of the `__hostlong` parameter.

See Also

`htons`, `ntohl`, `ntohs`

Example

Listing 7.30 - Using the `htonl` function

```

var
  ServerSocketID: integer;
  ServerAddressToBindTo: TsockAddrIn;

begin
  ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);

  //Define the Address to Connect to
  with ServerAddressToBindTo do
  begin
    sin_family := AF_INET;
    //listen to any IP address
    sin_addr.S_addr := htonl(INADDR_ANY);
    //listen on port 13
    sin_port := htons(13);
  end;
  ...
end;

```

htons *LibC.pas*

Syntax

```

function htons(
  __hostshort: uint16_t
):uint16_t;

```

Description

This function converts an unsigned 16-bit word value to a word value that is a network byte ordered value suitable for communicating across a network. This function is normally used when defining a port number to be stored in a `TsockAddr` structure. For more information on network byte ordering, see the section titled “Socket Addresses” in this chapter.

Parameters

`__hostshort`: This is the short integer (word) that is currently not a network byte ordered value.

Return Value

The function returns the network byte order value of the value passed into the `__hostshort` function.

See Also

htonl, ntohl, ntohs

Example**Listing 7.31 - Using the htons function**

```

var
  SocketAddress: TSocketAddr;
begin
  with SocketAddress do
  begin
    sin_family := AF_INET;
    sin_addr.S_addr := inet_addr('127.0.0.1');
    //Connect to port 80.
    sin_port := htons(80);
  end;
  ...
end;

```

inet_addr LibC.pas**Syntax**

```

function inet_addr(
  __cp: PChar
): in_addr_t;

```

Description

The `inet_addr` function converts an IPv4 address in the form 203.11.32.41 to an `in_addr_t` record, which can be used to pass into an address structure when defining which machine to connect to.

Parameters

`__cp`: The `__cp` parameter is a null-terminated string in the IPv4 address format that will be converted to a valid `in_addr_t` structure, such as 203.16.80.50.

Return Value

If the IP address passed into the `__cp` parameter is valid, the function will return a valid `in_addr_t` record containing the corresponding value of an IPv4 address so that it can be passed over the network. If the value passed into the `__cp` parameter is not a valid IP address, the function will return -1.

See Also

inet_ntoa

Example

See Listing 7.4 - Defining an address for use on the network.

inet_ntoa LibC.pas**Syntax**

```
function inet_ntoa(
  __in: in_addr
): PChar;
```

Description

The `inet_ntoa` function converts an `in_addr` structure that defines the host address given in network byte order to a value in standard IPv4 number format.

Parameters

`__in`: This is the address of the host in network byte formatted value.

Return Value

When successful, this function returns a pointer to a null-terminated string containing the IPv4 address of the address passed to the `__in` parameter. The value that is used is held within memory and should be immediately copied as other calls to the `inet_ntoa` function can change the data that the result will point to.

See Also

`inet_addr`

Example

See Listing 7.32 - Using the `ntohs` function.

listen LibC.pas**Syntax**

```
function listen(
  __fd: TSocket;
  __n: Cardinal
):Integer;
```

Description

Once a socket has been created and has bound to an address, it then needs to listen for client socket connections on the socket. The `listen` function creates a queue on the server so that client sockets can connect to the server and also defines the number of client connections that can connect to the socket before the clients begin being rejected by the socket server.

Parameters

`__fd`: This is the file descriptor of the socket that will be used as a server socket. This parameter is normally the return value of a successful call to the `socket` function.

`__n`: This parameter is the size of the backlog of client connections. Once the size of the queue is reached, clients will not be accepted to connect to the server socket.

Return Value

When successful, this function returns 0. If the function fails, it returns `-1`.

See Also

`socket`, `bind`, `accept`

Example

See Listing 7.11 - Using the `listen` function.

ntohl ***LibC.pas***

Syntax

```
function ntohl(  
    __netlong: uint32_t  
):uint32_t;
```

Description

This function converts a network byte order value integer value to an integer value that is valid on the host system. For more information on network byte ordering, see the section titled “Socket Addresses” in this chapter.

Parameters

`__netlong`: The `__netlong` parameter is the network byte ordered integer value to convert.

Return Value

The function returns an integer value that is byte ordered in the same way the host system is ordered.

See Also

`htonl`, `htons`, `ntohs`

ntohs ***LibC.pas***

Syntax

```
function ntohs(  
    __netshort: uint16_t  
):uint16_t;
```

Description

This function converts a network byte ordered word value that is typically used for a network port address to a value that is compatible on the host system. For more information on network byte ordering, see the section titled “Socket Addresses” in this chapter.

Parameters

`__netshort`: The `__netshort` parameter is a network byte ordered word value.

Return Value

The function returns a word value that is byte ordered in the same way the host system is ordered.

See Also

`htonl`, `htons`, `ntohl`

Example

Listing 7.32 - Using the `ntohs` function

```
var
  ServerSocketID: TFileDescriptor;
  NewClientSocketConnection: TFileDescriptor;
  ClientAddressInfo: TSocketAddr;
  ClientSocketLength: Cardinal;

begin
  ..
  ClientSocketLength := sizeof(ClientAddressInfo);
  NewClientSocketConnection := accept(ServerSocketID, @ClientAddressInfo,
    @ClientSocketLength);

  writeln('New Client Socket Connection #', NewClientSocketConnection);
  writeln('  IP Address: ', inet_ntoa(ClientAddressInfo.sin_addr));
  writeln('  Port: ', ntohs(ClientAddressInfo.sin_port));
  ...
end;
```

recv LibC.pas

Syntax

```
function recv(
  __fd: TSocket;
  var __buf;
  __n: size_t;
  __flags: Integer
):Integer;
```

Description

The `recv` function reads data from a socket. This function is similar to the `__read` function; however, the `recv` function allows for different options while reading the data from the socket. The available options for reading from a socket include reading out-of-band data and reading data from the socket while not removing the data from the socket's queue.

Parameters

`__fd`: This parameter is the file descriptor of the socket that will be read from.

`__buf`: This parameter is a buffer of data which will store the data read from the socket. The size of the data in this buffer should be at least the number of bytes in the `__n` parameter.

`__n`: The `__n` parameter defines the number of bytes to read from the socket.

`__flags`: For reading operations, the available options are:

`MSG_OOB`: Including this option will make the call to `recv` read out-of-band data from the socket.

`MSG_PEEK`: Including this option will read the data from the socket, but not remove it from the socket queue.

`MSG_WAITALL`: This option will block until `__n` bytes have been read from the socket, a signal is sent to the process, or the socket has disconnected.

`MSG_NO_SIGNAL`: Including this option will cause the `SIGPIPE` signal not to be sent to the process when the socket disconnects.

Return Value

When the `recv` function is successful, it returns the number of bytes that were read from the socket queue. If the function is unsuccessful, it returns `-1` and the error code can be found by calling the `errno` function.

See Also

`recvfrom`, `send`, `sendto`

Example

See Listing 7.14 - A simple HTTP socket server that handles client requests by forking.

recvfrom ***LibC.pas***

Syntax

```
function recvfrom(
  __fd: TSocket;
  var __buf;
  var __n: size_t;
  __flags: Integer;
  __addr: PSockAddr;
  __addr_len: PSocketLength;
):Integer;
```

Description

The `recvfrom` function retrieves data from a socket and also retains the location from which the data was sent. This function is normally used for datagrams when the datagram socket is receiving content and the address of where the message is sent from. When used with a connected socket, this function performs the same as the `send` function.

Parameters

`__fd`: This is the file descriptor of the socket.

`__buf`: This parameter is a pointer to a block of memory that will store the data read in from the socket. The size of the data in this buffer should be at least the number of bytes in the `__n` parameter.

`__n`: This parameter defines the size of the buffer in the `__buf` parameter in bytes.

`__flags`: For reading operations, the available options are:

`MSG_OOB`: Including this option will make the call to `recvfrom` read out-of-band data from the socket.

`MSG_PEEK`: Including this option will read the data from the socket, but not remove it from the socket queue.

`MSG_WAITALL`: This option will block until `__n` bytes have been read from the socket, a signal is sent to the process, or the socket has disconnected.

`MSG_NO_SIGNAL`: Including this option will cause the `SIGPIPE` signal not to be sent to the process when the socket disconnects.

`__addr`: This parameter is a pointer to a `TSockAddr` record that after a successful function call will hold the address from which the message was sent. This parameter can also be `nil` to indicate that the address does not need to be stored.

`__addr_len`: This parameter is a pointer to a cardinal that holds the size of the structure in the `__addr` function. After the function returns, this parameter will point to the actual size of the structure in the `__addr` parameter.

Return Value

When successful, this function returns the number of bytes read into the buffer. If this function is unsuccessful, it returns `-1` and the error code can be retrieved by a call to the `errno` function.

See Also

`recv`, `send`, `sendto`

Example

See Listing 7.34 - Using the `sendto` function.

select *LibC.pas*

Syntax

```
function select(  
  __nfd: Integer;  
  __readfds: PFDSet;  
  __writefds: PFDSet;  
  __exceptfds: PFDSet;  
  __timeout: Ptimeval;  
):Integer;
```

Description

The `select` function will block the current process and analyze a group of file sets to determine if the file descriptors have changed in any way. It will wait and see if the file descriptors contained within the `__readfds` set have data to read, if the file descriptors in the file descriptor set `__writefds` have written any data, and if any of the file descriptors in the `__exceptfds` group have caused any errors. The function also has an option to use a timeout value if no operation is performed within this time.

This function is normally used to watch a collection of client sockets for any notification change. Items can be manipulated in the sets by using the `FD_CLR`, `FD_ZERO`, `FD_ISSET`, and `FD_SET` functions. `FD_ZERO` will empty a file descriptor set of all entries, `FD_SET` will add a file descriptor to a file descriptor set, the `FD_CLR` function will remove a file descriptor from a file descriptor set, and the `FD_ISSET` function will determine if a file descriptor is contained within a file descriptor set.

Parameters

`__nfds`: This parameter contains the maximum file descriptor plus 1 number in any of the file descriptor sets.

`__readfds`: This parameter is a file descriptor set that will be watched for any new data that can be read by any of the file descriptors in the file descriptor set. If this parameter is `nil`, this data is not watched for any change.

`__writefds`: This parameter is a file descriptor set that will be watched for any data that indicates write operations have been performed on the file descriptors in the set. If this parameter is `nil`, this data is not watched for any change.

`__exceptfds`: This parameter is a file descriptor set that will be watched for any errors that happen within the file descriptor set. If this parameter is `nil`, then this data is not watched for any error.

`__timeout`: This parameter is a pointer to a `TTimeVal` parameter that defines how long the `select` function should wait for any notable change in the `__readfds`, `__writefds`, and `__exceptfds` fields. If this value is `nil`, then the function will wait forever. The `TTimeVal` data type contains two fields: the `tv_sec` field, which represents the included seconds of the timeout, and the `tv_usec` field, which represents the included milliseconds of the timeout.

```
type
    timeval = record
        tv_sec: __time_t;
        tv_usec: __suseconds_t;
    end;
    TTimeVal = timeval;
    PTimeVal = ^TTimeVal;
```

Return Value

If the function times out, it returns 0. If an error occurred within the function, it returns `-1`. If the function returned as a result of a change in the file descriptor set, it will return the total number of file descriptors in the file descriptor set.

*Example***Listing 7.33 - Using the select function**

```

program TriathlonRaceServer;

(*
    This is a console application that contains information about how particular
    contestants are doing within a triathlon race. Each contestant is given a
    time for each discipline (swim, ride, and run) and an athlete number.

    This is not a fully developed app. There should be things such as storing
    values in a database, but this is mainly used to demonstrate the select
    function.
*)

{$APPTYPE CONSOLE}

uses
    Libc, Classes, SysUtils, IniFiles, Contnrs, LibcHelperFunctions;

type
    TTriathlete = class
        AthleteNumber: integer;
        SwimTime: string;
        CycleTime: string;
        RunTime: string;
    end;

    TTriathletes = class(TObjectList)
    private
        function GetTriathlete(Index: integer): TTriathlete;
    public
        property Triathlete[Index: integer]: TTriathlete read GetTriathlete;
        function TriathleteByNumber(AthNo: integer): TTriathlete;
    end;

{ TTriathletes }

function TTriathletes.GetTriathlete(Index: integer): TTriathlete;
begin
    Result := Items[Index] as TTriathlete;
end;

function TTriathletes.TriathleteByNumber(AthNo: integer): TTriathlete;
var
    counter: integer;
begin
    Result := nil;
    for counter := 0 to Count - 1 do
        if Triathlete[counter].AthleteNumber = AthNo then
            begin
                Result := Triathlete[counter];
                break;
            end;
    end;
end;
end;

```

```

var
    Triathletes: TTriathletes;

    //Socket file descriptors used in the project
    ServerSocketID: TFileDescriptor = 0;
    NewClientSocketConnection: TFileDescriptor;

    //Socket information
    serverAddressDetails: TSockAddrIn;
    ClientAddressInfo: TSockAddr;
    ClientSocketLength: Cardinal;

    //Server sets used when listening for new connections
    ServerSocketSet: TFDSet;
    CopyOfServerSocketSet: TFDSet;

    counter: integer;
    BytesRead: integer;

procedure ReadRaceTimeEntryFromSocket(AFileDescriptor: integer);
var
    pContentLine: array[0..50] of char;
    DataString: string;
    iBytesRead: integer;
    Athlete: TTriathlete;
    TimeInfo: TStringList;
begin
    //This function reads a line from the file and then processes the string
    //the format will be
    //athleteNumber, Discipline, Time
    //where the Discipline will either be the value swim, cycle, or ride
    //The string will never be any more than 50 characters

    iBytesRead := recv(AFileDescriptor, pContentLine, 50, 0);
    if iBytesRead > 0 then
    begin
        DataString := Copy(StrPas(pContentLine), 1, iBytesRead);
        TimeInfo := TStringList.Create;
        try
            TimeInfo.CommaText := DataString;
            if TimeInfo.Count < 3 then
                Exit
            else begin
                //we have a Valid entry. Let's add the information
                Athlete := Triathletes.TriathleteByNumber(strtoint(TimeInfo[0]));
                if Athlete = nil then
                begin
                    //We need to add the athlete's information
                    Athlete := TTriathlete.Create;
                    Athlete.AthleteNumber := strtoint(TimeInfo[0]);
                    Triathletes.Add(Athlete);
                end;

                //Let's add the time split
                if TimeInfo[1] = 'swim' then
                    Athlete.SwimTime := TimeInfo[2]
            end;
        except
            TimeInfo.Free;
        end;
    end;
end;

```

```

        else if TimeInfo[1] = 'cycle' then
            Athlete.CycleTime := TimeInfo[2]
        else if TimeInfo[1] = 'run' then
            Athlete.RunTime := TimeInfo[2];

        //Let's display the current info on the athlete
        writeln('Split Info Received: ', DataString);
        with Athlete do
            writeln(Format('Athelete #d, Swim (%s), Cycle (%s), Run (%s)',
                [AthleteNumber, SwimTime, CycleTime, RunTime]));
        end;
    finally
        TimeInfo.Free;
    end;
end;
end;

{ The main function that handles all the processing of incoming requests }

begin
    Triathletes := TTriathletes.Create(true);
    try
        //The first thing that we do is set up the socket connection
        //Connect to the socket
        ServerSocketID := socket(AF_INET, SOCK_STREAM, 0);

        if ServerSocketID = -1 then
            begin
                //Could not create a new socket
                perror('Could not create a socket');
                Exit;
            end;

        with serverAddressDetails do
            begin
                sin_family := AF_INET;
                sin_addr.S_addr := htonl(INADDR_ANY); //Listen on any address
                sin_port := htons(2201); //Listen on Port 2201
            end;

        if bind(ServerSocketID, serverAddressDetails, sizeof(serverAddressDetails)) = -1
        then
            begin
                //Could not bind to the requested address
                perror('Could not bind to the address');
                Exit;
            end;

        if listen(ServerSocketID, 20) = -1 then
            begin
                perror('Could not listen on the Socket');
                Exit;
            end;

        FD_ZERO(ServerSocketSet);
        FD_SET(ServerSocketID, ServerSocketSet);
    
```

```

writeln('Accepting Connections');
repeat
  CopyOfServerSocketSet := ServerSocketSet;

  //The select function will watch for a change in the file descriptors
  //and return when a change in the file descriptor has occurred.
  select(FD_SETSIZE, @CopyOfServerSocketSet, nil, nil, nil);

  //So we have a change in the file descriptors. Let's examine them
  //to see the change of socket connections.
  for counter := 0 to FD_SETSIZE - 1 do
    if FD_ISSET(counter, CopyOfServerSocketSet) then
      begin
        //We have a change in the status of our file descriptors
        if counter = ServerSocketID then
          begin
            //A new friend has come to connect with us.
            ClientSocketLength := sizeof(ClientAddressInfo);
            NewClientSocketConnection := accept(ServerSocketID, @ClientAddressInfo,
              @ClientSocketLength);
            FD_SET(NewClientSocketConnection, ServerSocketSet);
            writeln('New Client Socket Connection #', NewClientSocketConnection,
              ' ('+inet_ntoa(ClientAddressInfo.sin_addr),
              ':'+ntohs(ClientAddressInfo.sin_port));

            //We should get the details and write the time entry into the athlete's
            //list
            ReadRaceTimeEntryFromSocket(NewClientSocketConnection);
          end else begin
            //The connection change is from an existing client
            //This may be the result of a disconnection
            //The constant FIONREAD is declared in the LibcHelperFunction unit
            //that ships with the book.
            ioctl(counter, FIONREAD, @BytesRead);

            if BytesRead = 0 then
              begin
                //The client has decided to leave us
                writeln('Socket Disconnected. File Descriptor: ', counter);
                __close(counter);
                FD_CLR(counter, ServerSocketSet);
              end else begin
                //Read from the buffer. This will be caused by a change in status only
                end;
              end;
            end;
          until false; //Loop indefinitely
        finally
          Triathletes.Free;
          __close(ServerSocketID);
          writeln('Triathlon Results Server is closing down.');
```


sethostent LibC.pas**Syntax**

```
procedure sethostent(
  __stay_open: Integer
);
```

Description

The sethostent function opens the host's database so that the host's entries can be read in using the gethostent function. Apart from resetting the reading position to the start of the host's database, this function also specifies whether or not the connection to the host's database should remain open for more efficient access.

Parameters

__stay_open: When this parameter is set to a non-zero value, the connection to the host's database will remain open so that calls to gethostent will be faster.

See Also

gethostent, endhostent

Example

See Listing 7.21 - Using the gethostent function.

setservent LibC.pas**Syntax**

```
procedure setservent(
  __stay_open: Integer
);
```

Description

The setservent function opens the /etc/services file so that service entries from the file can be read into the TServEnt record structure using the getservent function. If entries have already been read in from the /etc/services file, calling this function will rewind the read position to the first service. This function is similar in concept to opening a file, but this function works specifically with service entries from the /etc/services file.

Parameters

__stay_open: When this value is 1, the /etc/services file will not be closed if a call to getservbyname or getservbyport is used. If the value 0 is used, the file will be closed when getservbyname or getservbyport is used.

See Also

getservbyname, getservbyport, getservent, endservent

Example

See Listing 7.27 - Using the getservent function.

send LibC.pas

Syntax

```

function send(
  __fd: TSocket;
  const __buf;
  __n: size_t;
  __flags: Integer
):Integer;

```

Description

The send function transmits data to a connection-based socket, such as a socket created with the SOCK_STREAM option. The function also allows specific options for the socket, such as removing the socket's ability to route between networks, sending data without blocking the process, sending out-of-band data, and allowing the process not to receive signals as a result of a broken socket connection. If no options are used for the socket, the __write function can be used in its place.

Parameters

__fd: This is the file descriptor of the socket where data will be sent.

__buf: The __buf parameter points to a buffer of data that will be written to the socket.

__n: The __n parameter is the size of the memory buffer the __buf parameter points to in bytes.

__flags: This parameter contains a list of options that can be used when sending data to the socket. The available options to include in the __flags parameter are:

MSG_OOB: This will allow the data to be sent out of band. Out-of-band data is data that is flagged as urgent and usually indicates that the server the socket is connecting to does not have the space on the queue to accept the connection.

MSG_DONTROUTE: Including this option will result in the data being sent only through the local network and not through the router by using the machine's default gateway.

MSG_DONTWAIT: Using this option will result in the function returning immediately. If the data cannot be sent, the function will result in error and a call to the errno function will return EAGAIN.

MSG_NOSIGNAL: Including this option will mean that the SIGPIPE signal will not be sent to the current process if the connection to the socket is terminated.

Return Value

If the function is successful, it returns the number of bytes written to the socket. If the function fails, it will return -1.

See Also

sendto, __write

Example

See Listing 7.13 - The complete code for the DayTime server.

sendto LibC.pas*Syntax*

```
function sendto(
  __fd: TSocket;
  const __buf;
  const __n: size_t;
  __flags: Integer;
  const __addr: sockaddr;
  __addr_len: socklen_t
):Integer;
```

Description

The sendto function is used to send data to another socket at a specific address. The sendto function can send data to both a connection-based protocol, such as TCP, and disconnected sockets using UDP, although the function is mainly used for UDP communication. The function will send data similarly to using the send function, but this function specifies the address where the data will be sent.

Parameters

__fd: This is the file descriptor of the socket that was returned from the socket function.

__buf: This is a pointer to a buffer that holds the data that will be sent to the socket.

__n: This is the number of bytes of the buffer that will be sent to the socket.

__flags: This parameter contains a list of options that can be used when sending data to the socket. The available options to include in the __flags parameter are:

MSG_OOB: This will allow the data to be sent out of band. Out-of-band data is data that is flagged as urgent and usually indicates that the server the socket is connecting to does not have the space on the queue to accept the connection.

MSG_DONTROUTE: Including this option will result in the data being sent only through the local network and not through the router by using the machine's default gateway.

MSG_DONTWAIT: Using this option will result in the function returning immediately. If the data cannot be sent, the function will result in an error and a call to the errno function will return EAGAIN.

MSG_NOSIGNAL: Including this option will mean that the SIGPIPE signal will not be sent to the current process if the connection to the socket is terminated.

`__addr`: This parameter is a `TSockAddr` variable that holds the address of the location to which the data will be sent.

`__addr_len`: The `__addr_len` parameter is the size of the address structure passed into the `__addr` function and is usually `sizeof(TSockAddr)`.

Return Value

When successful, this function will return the number of bytes that were sent to the socket. If the function failed, it will return `-1` and the error code for the function can be found by using the `errno` function.

See Also

`send`, `recv`, `recvfrom`

Example

Listing 7.34 - Using the `sendto` function

```
//Code to send a stock name/price to a datagram server
procedure TfrmStockClient.btnCheckItClick(Sender: TObject);
var
  ServerAddress: TSockAddr;
  SocketFD: integer;
  StockName, IPAddress: array[0..50] of char;
  StockItem: TListItem;
begin
  //This function sends the stock data to the stock server in a datagram.

  //Define the Address.
  with ServerAddress do
  begin
    sin_family := AF_INET;
    sin_port := htons(strtoint(edServerPort.Text));
    StrPCopy(IPAddress, edServerIP.Text);
    sin_addr.S_addr := inet_addr(IPAddress);
  end;

  //Create the socket datagram
  SocketFD := socket(AF_INET, SOCK_DGRAM, 0);

  if SocketFD = -1 then
  begin
    MessageDlg('The Socket could not be created.', mtInformation, [mbOk], 0);
    Exit;
  end;

  try
    //Send the stock name and price to the server
    StrPCopy(StockName, cbStock.Text + ':' + edStockPrice.Text);
    sendto(SocketFD, StockName, StrLen(StockName), 0, ServerAddress,
           sizeof(ServerAddress));

    //Add the stock to the list View, or update if it's already there
    StockItem := lvStocks.FindCaption(0, cbStock.Text, false, true, true);
    if StockItem = nil then
```

```

        begin
            StockItem := lvStocks.Items.Add;
            StockItem.Caption := cbStock.Text;
            StockItem.SubItems.Add(edStockPrice.Text);
        end else
            StockItem.SubItems[0] := edStockPrice.Text;
        finally
            _close(SocketFD);
        end;
    end;

program SimpleStockServer;

{$APPTYPE CONSOLE}

(*
    This program is a datagram (UDP) server that handles simple connections
    by gathering new information on a stock price and stores and displays them.

    This demo doesn't work with a database to store the stock
    information. It simply displays the price on-screen. It is used merely to
    demonstrate using datagrams.
*)

uses
    SysUtils, Libc;

var
    ServerSocket: TFileDescriptor;
    ServerAddress: TSocketAddr;
    DataBuffer: array[0..50] of char;
    StockData: string;
    StockName: string;
    StockPrice: string;
    IsOK: integer;

begin
    writeln('Starting the Stock Listening Server.');
```

//Let's create the socket

```

    ServerSocket := socket(AF_INET, SOCK_DGRAM, 0);
    try
        //Let's define the address for the server
        with ServerAddress do
            begin
                sin_family := AF_INET;
                sin_port := htons(8876); //We will use this port for the server
                sin_addr.S_addr := inet_addr('127.0.0.23');
            end;

        //Let's bind to that address
        bind(ServerSocket, ServerAddress, sizeof(ServerAddress));

        writeln('The Server is now listening.');
```

//Now we can process requests for the stock

```

repeat
  IsOk := recvfrom(ServerSocket, DataBuffer, 50, 0, nil, nil);

  if IsOk = -1 then
    begin
      perror('Error Recieving');
      Exit;
    end else begin
      //Let's disassemble the Stock Information. The details should be
      //in the format. "StockName: Price". For example REDHAT:203.55
      StockData := Copy(StrPas(DataBuffer), 1, IsOk);
      StockName := Copy(StockData, 1, Pos(':', StockData)-1);
      StockPrice := Copy(StockData, Pos(':', StockData)+1, Length(StockData));
      writeln('Stock Data Recieved. ');
      writeln('  Stock: ', StockName);
      writeln('  Price: ', StockPrice);
    end;
  until false;
finally
  __close(ServerSocket);
  writeln('Closing the Server');
end;
end.

```

setsockopt *LibC.pas*

Syntax

```

function setsockopt(
  __fd: TSocket;
  __level: Integer;
  __optname: Integer;
  __optval: Pointer;
  __optlen: socklen_t
):Integer;

```

Description

The `setsockopt` function defines options for an existing socket such as the size of the send and receive buffers, socket lifetime, routing options, and others.

Parameters

`__fd`: This parameter is the file descriptor of a valid socket.

`__level`: This is the level of the socket option, and represents the category of option, such as if you want to examine the socket options or TCP options. The Kylix implementation of `LibC.pas` only includes the socket option level `SOL_SOCKET`, which is used for examining socket options. Although there are other available values for this parameter, they are not declared in Kylix, such as specific options for TCP.

`__optname`: This is the option that will be set. When setting socket options, the available options are:

SO_REUSEADDR: This option defines if another server can bind to a socket on an address even if a process is handling a client on that address, but the original server has closed.

SO_KEEPALIVE: Using this option will result in the socket that it is connected to sending messages on a regular basis to ensure that the connection is still active.

SO_LINGER: This option sets the outcome of what will happen to a reliable socket when closed if data is still waiting to be sent from that socket.

SO_BROADCAST: This option specifies whether UDP broadcasts are possible from this socket. Broadcasting is turned off by default in all sockets.

SO_OOBINLINE: This specifies if out-of-band data can be read in by the application in the same way normal data is read, using the `__write` or `recv` functions.

SO_SNDBUF: This option sets the size of the send buffer of the socket.

SO_RCVBUF: This option sets the size of the receive buffer of the socket.

SO_DONTROUTE: Setting this option specifies whether or not the socket is allowed to route out of the local network.

__optval: The `__optval` parameter is a pointer to the value that will be used for setting the socket option.

__optlen: The `__optlen` parameter defines the size of the value that the `__optval` parameter points to.

Return Value

With a successful function call, the function returns 0. When the function is unsuccessful, it returns `-1` and the error code can be discovered by calling the `errno` function.

See Also

`getsockopt`

Example

Listing 7.35 - Using the `setsockopt` function

```
//We don't need such a large receiver buffer for a DayTime client
ReadBufferSize := 80;
setsockopt(SocketFileDescriptor, SOL_SOCKET, SO_RCVBUF,
  @ReadBufferSize, sizeof(integer));
```

shutdown *LibC.pas*

Syntax

```
function shutdown(
  __fd: TSocket;
  __how: Integer
):Integer;
```

Description

The shutdown function closes down part or all of the read and write functionality of a current socket. It has the ability to not allow the socket to read, not allow the socket to write, or not allow the socket to do either.

Parameters

`__fd`: The `__fd` parameter represents the socket that will have its read/write facilities limited.

`__how`: The `__how` parameter defines how the socket is to be limited and will either be `SHUT_RD` to not allow data to be received on the socket, `SHUT_WR` to not allow data to be written to the socket, or `SHUT_RDWR` to not allow data to be received or written to the socket.

```
const
    SHUT_RD = 0;
    SHUT_WR = 1;
    SHUT_RDWR = 2;
```

Return Value

If the function is successful, it returns 0. If the function fails, it returns -1. The function will normally only fail if the `__fd` parameter is not the file descriptor of a valid socket or if the `__how` parameter is not one of the values `SHUT_RD`, `SHUT_WR`, or `SHUT_RDWR`.

See Also

socket, socketpair

socket LibC.pas

Syntax

```
function socket(
    __domain: Integer;
    __type: __socket_type;
    __protocol: Integer
): TSocket;
```

Description

The socket function creates a socket that is used as an endpoint of either a client or server of the connection. The function specifies the protocol family, the type of socket, and options for the socket. For more information on creating a socket, see the section titled “Creating a Socket” in this chapter.

Parameters

`__domain`: The `__domain` parameter defines the addressing mechanism that will be used for the socket. The available options for this parameter are:

`PF_LOCAL`, `PF_UNIX`, `PF_FILE`: Any of these options are valid for local UNIX sockets.

PF_INET: This function is used for the Internet IPv4 socket family.

PF_AX25: The protocol used for amateur radio.

PF_IPX: Novell's IPX protocol family.

PF_APPLETALK: The AppleTalk protocol.

PF_INET6: This function is used for the Internet IPv6 socket family.

`__type`: The `__type` parameter defines the communication type of the socket. Only one value can be chosen from `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`.

`SOCK_STREAM`: This is for communication using Transmission Control Protocol (TCP).

`SOCK_DGRAM`: The socket uses the User Datagram Protocol (UDP).

`SOCK_RAW`: The socket uses raw sockets to send/receive low-level network socket data.

`__protocol`: This parameter defines the options for the socket. The available options are the same options listed in the `setsockopt` function. To use the default values for the socket family and type combination, use the value 0.

Return Value

If a socket can be successfully created, the function will return the file descriptor of a created socket. If the function is not successful, which is usually the result of an incorrect combination of parameters, it will return `-1`.

See Also

`socketpair`

Example

See Listing 7.1 - Creating a socket.

socketpair *LibC.pas*

Syntax

```
function socketpair(
  __domain: Integer;
  __type: __socket_type;
  __protocol: Integer;
  var __fds: TSocketPair
):Integer;
```

Description

The `socketpair` function creates a pair of socket file descriptors that can be used for reading and writing to a socket. This function is similar to the `pipe` function, but where the `pipe` function will return two file descriptors, one for reading and the other for writing, the `socketpair` function will actually return two file descriptors, which both read and write. Each file descriptor has the other file descriptor as its source for reading and destination for

sending. Although this function is a socket function, it really is more of a function geared towards interprocess communication as it will only work with local sockets.

Parameters

`__domain`: The `__domain` parameter defines the addressing mechanism that will be used for the socket. The only available options for this parameter are `PF_LOCAL`, `PF_UNIX`, or `PF_FILE`.

`__type`: The `__type` parameter defines the communication type of the socket. Only one value can be chosen from `SOCK_STREAM` or `SOCK_DGRAM`.

`SOCK_STREAM`: This is for communication using Transmission Control Protocol (TCP).

`SOCK_DGRAM`: The socket uses the User Datagram Protocol (UDP).

`__protocol`: This parameter defines the options for the socket. When using the `socketpair` function, only the value 0 is valid.

`__fds`: This parameter is a variable that is an array of `TSocket` with two elements. After this function is called, the socket file descriptors will be contained within the variable. The file descriptors that are used have each other as the default destination for the communication.

Return Value

When successful, this function returns 0 and the `__fds` parameter will contain file descriptors of the socket. If the function fails, it will return -1 and a call to the `errno` function will return the error code.

See Also

socket, pipe

Example

Listing 7.36 - Using the `socketpair` function

```
program socketPairExample;

{$APPTYPE CONSOLE}

//This demonstration is similar to the pipe demo from Chapter 5,
//but instead this example uses socketpairs instead of pipes

uses
  Libc;

type
  TCustomer = record
    Name: array[0..30] of char;
    Age: integer;
    Email: array[0..45] of char;
  end;

var
  Sockets: TSocketPair;
```

```

intProcess: integer;
SomeCustomer: TCustomer;
intDataLength: integer;
i: integer;

begin
  writeln('Starting the socketpair demo.');
```

//Create the socket Pair

```

  if socketpair(AF_LOCAL, SOCK_STREAM, 0, Sockets) = -1 then
    begin
      writeln('The socketpair could not be created.');
```

Exit;

```

    end;

  //Let's fork the process
  intProcess := fork;
  try
    if intProcess = -1 then
      begin
        //There was a problem forking the process
        writeln('There was a problem forking the process...');
```

end else if intProcess = 0 then

```

    begin
      //We are the parent process. Let's write some stuff to the pipe.
      writeln(getpid, ': The parent is now writing to the socketpair');
```

//Let's write a few customers to the pipe

```

    with SomeCustomer do
      begin
        Name := 'John Doe';
        Age := 25;
        Email := 'johndoe@notreal.com';
      end;
      send(Sockets[0], SomeCustomer, sizeof(TCustomer), 0);

      with SomeCustomer do
        begin
          Name := 'Jane Doe';
          Age := 23;
          Email := 'janedoe@notreal.com';
        end;
        send(Sockets[0], SomeCustomer, sizeof(TCustomer), 0);
    end else begin
      //We are the child process
      writeln(getpid, ': The child process is now reading from the socketpair');
```

//Let's read the entries from the Pipe, one record at a time.

```

    for i := 1 to 2 do
      begin
        intDataLength := recv(Sockets[1], SomeCustomer, sizeof(TCustomer), 0);
        if intDataLength <> 0 then
          begin
            //Let's display the information about the customer
            writeln(getpid, ': [Name]: ', SomeCustomer.Name, ' [Age]: ',
              SomeCustomer.Age,
              ' [Email]: ', SomeCustomer.Email);
          end;
        end;
      end;
    end;
  end;
end;
```

```
        end;  
    end;  
finally  
    __close(Sockets[0]);  
    __close(Sockets[1]);  
end;  
end.
```

Where to Go From Here

In this chapter, you have learned how to create individual socket clients and servers for Linux and how to take advantage of the Internet daemons that are part of the Linux system that you use today. You have also learned how to obtain the socket file descriptors from a TClientSocket or TServerSocket component so that you can mix NetCLX and Linux API function calls.

You should find that with the knowledge you have now, you can implement a socket server to fulfill almost any purpose with the Linux API. If you understand the NetCLX components that ship with Kylix, your socket programming will be unstoppable.

Importing Other APIs

Introduction

One of my favorite sayings is “give a man a fish and feed him for a day, teach him how to fish and feed him for a lifetime.” Although Borland programmers have done a fantastic job converting the Linux API, there are many other APIs that have not been converted to their Object Pascal equivalents because they were written by proprietary companies, open source groups, or individuals, and it is unreasonable to expect that Borland will convert every Linux API that comes along.

In this chapter we will teach you to fish with APIs, meaning you will learn the basics of converting an existing C header file or an existing shared object written in Kylix or C so that it can be used within your application. You will also learn how to call a shared object dynamically without knowing until run time where the shared object is located.

Getting Starting with Shared Objects

Shared objects are a collection of functions that are stored in a separate file that can be linked to from an application, and the application can call the functions contained within the shared library, just as if it were calling functions compiled into the application.

To put these kinds of functions into perspective, the entire Linux API is actually stored with many shared object files such as `libc.so.6`, `libpthread.so.0`, `libcrypt.so.1`, and several other files. Under Linux, shared object files most often live in the `/lib` or `/usr/lib` directories or any directory that exists in the `LD_LIBRARY_PATH` environment variable. In many cases you may find that the shared object file does not live in one of these directories but instead in a linked file, which points to where the actual shared object file is. These linked files often include version details of the shared object, so that the correct version of the library can be called if needed, or if a new version of the shared library is created then a linked file with the generic version number will point to the latest version.

An example of this would be the OpenGL libraries that handle three-dimensional drawing for Linux. The actual shared object file that contains all the functions on my local system is `LibGL.so.1.2.030300`, but linked files called `LibGL.so` and `LibGL.so.1` are in the same directory.

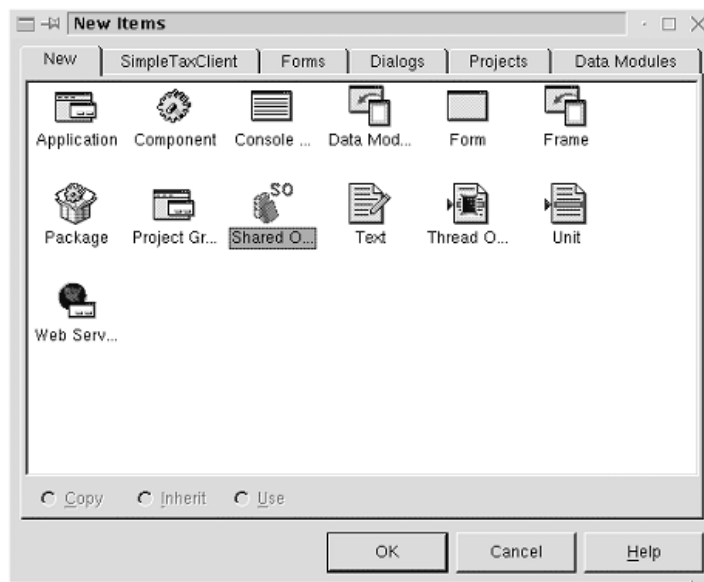
Now that you know the basics of shared objects with Kylix, let’s look at how a shared object is made with Kylix.

Creating a Shared Object with Kylix

When you create a shared object with Kylix, you are creating a Kylix project. The only difference between a Kylix application and a Kylix shared object is the keyword at the start of a file. For an application the keyword used is `program`, as shown in Listing 8.1. For a shared object, the keyword `library` is used, as shown in Listing 8.2.

To create a new shared object project from the Kylix IDE, simply choose `File|New` from the Kylix menu as shown in Figure 8.1 and then select `Shared Object`. After you have made the selection, you can add the functions that you want included within your shared object and the functions that you want to export from the shared object. Once you have created your shared object you compile the application and then install the shared object by copying it to a directory contained within the `LD_LIBRARY_PATH` environment variable or by creating a linked file in the same directory to wherever the shared object file is actually residing.

Figure 8.1 -
Creating a
new shared
object library



Listing 8.1 - An application

```
program ASimpleApplication;

uses
  SysUtils,
  Classes;

function GetTaxAmount(Amount: double): double;
begin
  //This tax is 10% of the amount
  Result := Amount * 0.10;
end;

function GetTaxName: string;
```

```

begin
    Result := 'Simple Tax System';
end;

procedure GetVersion(var MajVersion, MinVersion: integer);
begin
    MajVersion := 1;
    MinVersion := 0;
end;

begin
    writeln('The tax on $20 is ', GetTaxAmount(20));
    writeln('Tax: ', GetTaxName);
end.

```

Listing 8.2 - A shared object

```

library SimpleTaxSystem;

uses
    SysUtils,
    Classes;

function GetTaxAmount(Amount: double): double;
begin
    //This tax is 10% of the amount
    Result := Amount * 0.10;
end;

function GetTaxName: string;
begin
    Result := 'Simple Tax System';
end;

procedure GetVersion(var MajVersion, MinVersion: integer);
begin
    MajVersion := 1;
    MinVersion := 0;
end;

exports
    GetTaxAmount,
    GetTaxName,
    GetVersion;

begin
end.

```

As you can see in Listings 8.1 and 8.2, the structure of a shared object library is not that different from an application. The difference between the application and the shared object is that the application in this case has code that actually calls the functions contained within the application. The shared object, on the other hand, is used to store functions so that different applications can access them. To do that, we need to be able to call the functions from a separate application.

Using the Shared Object Created with Kylix

After you have compiled the shared object library and installed it, you should then go about setting up a client application.

To set up a client application to the shared object, you need a way to access the functions stored within the shared object library. Luckily Kylix has a simple way of doing just this. You can tell Kylix that a function is within a shared object file by declaring the function the same way you declared it within the shared object, but then adding the external declaration followed by the name of the shared object file and the name the function is called within the shared object. For example:

```
external 'filename.so' name 'myfunction';
```

The name declaration is optional and is used when the name you want called within your application is different from the name of the function in the shared object.

The code to declare the function from Listing 8.2 would then look like the code in Listing 8.3.

Listing 8.3 - Code to access a shared object written in Kylix

```
unit TaxClientRoutines;

{
    This unit imports the functions that calculate
    tax based on a simple structure
    Written for the Tomes of Kylix - The Linux API
    Published by Wordware
}

interface

const
    TaxLibraryName = 'libSimpleTaxSystem.so';

function GetTaxAmount(Amount: double): double; external TaxLibraryName;
function GetTaxName: string; external TaxLibraryName;
procedure GetVersion(var MaxVersion, MinVersion: integer); external TaxLibraryName;

implementation

end.
```

You could include the above functions in the unit of the form, but it is a better idea, when working with a shared object, to create a separate unit that deals specifically with that shared object. It is also a good idea and a Kylix coding convention when stating the filename of the shared object to declare the filename as a constant just in case the filename of the shared object changes.

Now that you have the code to call the functions you can do so in exactly the same way you would call normal functions as shown in Listing 8.4.

Listing 8.4 - An application that makes use of the shared object

```

unit unitTaxClientForm;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QstdCtrls, TaxClientRoutines;

type
  TfrmTaxSharedObjClient = class(TForm)
    Label1: TLabel;
    lblTaxName: TLabel;
    Label3: TLabel;
    lblTaxVersion: TLabel;
    Label5: TLabel;
    edAmount: TEdit;
    btnCalculateTax: TButton;
    lblResultingTax: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnCalculateTaxClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmTaxSharedObjClient: TfrmTaxSharedObjClient;

implementation

{$R *.xfm}

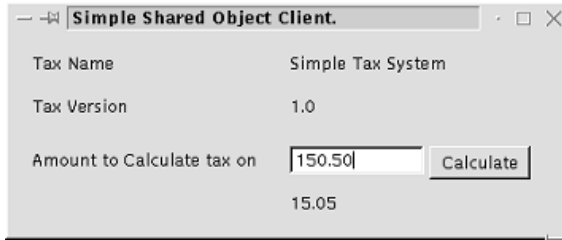
procedure TfrmTaxSharedObjClient.FormCreate(Sender: TObject);
var
  MajVersion, MinVersion: integer;
begin
  lblTaxName.Caption := GetTaxName;
  GetVersion(MajVersion, MinVersion);
  lblTaxVersion.Caption := Format('%d.%d', [MajVersion, MinVersion]);
end;

procedure TfrmTaxSharedObjClient.btnCalculateTaxClick(Sender: TObject);
begin
  lblResultingTax.Caption := CurrToStr(
    GetTaxAmount(StrToCurr(edAmount.Text)));
end;

end.

```

Figure 8.2 -
The shared
object client
application



Now that a client application for the shared object is written, you will be able to make use of separating the application from business rules. It is often a good idea when building applications to separate the business rules from the application. In this way, when you ship your application, if the business rules that are stored in the shared object change, you only need to reshipe the shared object as an update. This is an effective solution if you know that the business rules are likely to change.

Creating a Shared Object in C

Now that you've created your first shared object and an application that accesses the shared object, you may think that you know all you need to about working with shared objects. However, Kylix is a relative newcomer to the Linux development world. As Linux is written in C, most of the shared objects that you come across will also be written in C. As a result, it is important when importing other shared objects that you know a little about how C-written shared objects are made to understand how you can use the shared objects more efficiently.

So we will create a shared object file from C to demonstrate importing it into our application. The same functions we used in the Kylix shared object will be used, but we will modify the library so that it can be written in C and use standard C data types. The code for the shared object and its makefile can be found in Listing 8.5 and Listing 8.6, respectively.

Listing 8.5 - C code for a shared object file

```
#include <stdio.h>

extern double GetTaxAmount(double amount) {
    //This returns a third of the amount
    return amount / 3;
}

extern void GetVersion(int *majorVersion, int *minorVersion) {
    *majorVersion = 2;
    *minorVersion = 1;
}

extern void GetTaxName(char *TaxName) {
    strcpy(TaxName, "A C written Tax Library\0");
}
```

Listing 8.6 - Makefile for the C-written shared object

```
#
# Makefile for the shared object file to demonstrate creating a
# shared object file in C and then calling it from Kylix
#
# written for the "Tomes of Kylix - The Linux API" by Glenn Stephens
# published by Wordware Publishing www.wordware.com
#

AnotherTaxLibrary.so: AnotherTaxLibrary.c
    gcc -fPIC -c -g AnotherTaxLibrary.c -o AnotherTaxLibrary.o
    gcc -g -shared -Wl,-soname,AnotherTaxLibrary.so -o AnotherTaxLibrary.so.1
AnotherTaxLibrary.o -lc
```

If you are familiar with coding in C, the code in Listing 8.5 will seem fairly clear. If you are not quite familiar with C, then don't worry, we will work through some rules that will help you import the libraries quite easily. When importing functions from a shared object for which you only have the C definition, the following simple rules should help you in the majority of conversions.

Rules for Importing Functions from a C-Written Library

Rule 1. Understand the way parameters and return values work in functions in both C and Kylix.

When you look at a normal function with Kylix, you will have the structure:

```
function FunctionName(ParamName: ParameterType): ReturnType;
```

A C function would be declared a little differently as the following code demonstrates:

```
ReturnType FunctionName(ParameterType ParamName);
```

Also, if the `ReturnType` in the C function is "void", you are dealing with a procedure instead of a function.

You should also keep in mind that C is a case-sensitive language, so when you name the functions in Kylix, make sure that they match the capitalization of their C counterparts.

Rule 2. C uses a different calling convention than Kylix.

Under Kylix, when a function is compiled, the parameters are passed internally from left to right, using registers to pass the parameters and cleaning up the stack after the function returns. This calling convention is called *register* and is the default for all Kylix functions.

On the other hand, C uses the *cdecl* calling convention, so parameters are passed in from right to left without using registers. So this means that in your declaration of the function you should put the *cdecl* directive after the function call.

Rule 3. Kylix string types cannot be used. Use `PChar` instead.

The C world does not have a string data type. Instead, it uses an array of `char` to declare a string, so you cannot use the Pascal string data type in your definitions. Also, do not use Kylix strings in your Kylix-written shared objects if you want a C application to access your shared object.

Rule 4. Change C pointer types to pass-by-reference parameters or Pascal pointer types.

In C, a parameter type or return data type is a pointer prefixed with an asterisk character (*), as shown in the `GetVersion` function in Listing 8.5. Because this means that the parameter is a pointer, you have the option of declaring the parameter as a var parameter if the function will change the values of the parameter, as a const parameter if the function will not change the contents of the parameter, or as a pointer to that data type. In the `GetVersion` example, we should use the var option as the function will change the values.

Rule 5. Use the correct data type when converting parameters.

If you do not use the correct data type when converting from the C declaration to the Kylix equivalent, your function calls will not be accurate and this may cause your application to do some crazy things, as the function will think it has different parameters than what is expected. You need to be sure that the data types you choose are correct. If you cannot find a corresponding data type, the best way to work around the problem is to at least have a Kylix data type that is the same size as its C counterpart.

Table 8.1 lists the basic data type conversions from C to Kylix.

Table 8.1 - Data types in C and Kylix

The C data type	Is represented in Kylix by...
char*	PChar
char	char or byte
int	integer
double	double
word	word
float	single
long double	extended
void *	Pointer

Rule 6. Convert C #defines into constants.

The C language uses definitions for its constants. A constant declaration that sets a constant `MinimumSpending` to 100 in C would be:

```
#define MinimumSpending 100
```

where in Kylix, the equivalent would be the following:

```
const MinimumSpending = 100;
```

Rule 7. Convert C macros into functions.

Some definitions in a C header file may not be constants but may take any number of parameters. These are called macros, and to convert these to Kylix, you should convert them into valid functions. For example, a C macro that squares a number would be declared as:

```
#define SQUARETHENUMBER(num)    (num*num)
```

To convert this into Kylix, you would simply redeclare the macro as a function in the form of:

```
function SQUARETHENUMBER(num: double): double;
begin
```

```
Result := num * num;
end;
```

Rule 8. Change C structures to Object Pascal records.

A record in C is declared as a struct, so when you convert that into the Kylix equivalent, you should define a new record with the same field names and place the fields in the same order. To demonstrate this point, let's convert a structure defined in the joystick.h C header file to its corresponding Kylix equivalent.

The C definition of the record is:

```
struct JS_DATA_TYPE {
    int buttons;
    int x;
    int y;
};
```

and the Kylix conversion of the record is:

```
type
    JS_DATA_TYPE = record
        buttons: integer;
        x: integer;
        y: integer;
    end;
    TJoystickDataType = JS_DATA_TYPE;
```

You should notice that the case remains consistent between the record names. This is an important thing to remember when converting any C definition. You may also notice that a record alias called TJoystickDataType is created in the Kylix definition. This allows the code that uses that record structure to be more Kylix friendly.

Using a C-Created Shared Object in Your Application

Using the rules in the previous section, you should have enough knowledge to convert C functions into their Kylix counterparts. The functions you have should look something like the ones in Listing 8.7.

Listing 8.7 - A Kylix unit to call the functions from the C library

```
unit AnotherTaxLibraryFunctions;

interface

const
    TaxLibraryName = 'AnotherTaxLibrary.so.1';

function GetTaxAmount(Amount: double): double; cdecl; external TaxLibraryName;
procedure GetVersion(var MajorVersion, MinorVersion: integer); cdecl; external
    TaxLibraryName;
procedure GetTaxName(TaxName: PChar); cdecl; external TaxLibraryName;

implementation

end.
```

You should notice that Listing 8.7 follows the rules for converting the shared object file. For the `GetTaxAmount` function, the Kylix parameters match how they were declared in C and the function uses the `cdecl` calling convention. `GetVersion` within the library is declared as a void function, which means in fact it is a procedure. You will also notice that the C declaration uses a pointer type and its Kylix equivalent makes the parameter a var parameter, as the function actually changes the values of the parameters. And lastly, `GetTaxName` demonstrates the use of C string data types and how they should be converted to Kylix. Any C string will be passed in as a pointer to a character array, which is represented by the `PChar` data type in Kylix.

This section should give you enough knowledge to link to a shared object file from within your application, but you may have a situation where you will not know which file the library is stored in, or you may have many shared object libraries that create a plug-in system for your application so that your application can change dynamically depending on which library it uses. To handle these situations, you need to be able to dynamically load a shared object at run time.

Dynamically Loading a Shared Object

Not being tied to a particular filename allows you to create a plug-in framework for your application, allowing separate shared objects to behave in a polymorphic nature. Having a framework as a group of shared objects also allows developers of languages other than Kylix to develop plug-ins for your application. We have seen this already with the tax system, so we will modify the system we have in place to dynamically call any shared object that implements the same functions in the same way.

To access a shared object function dynamically, you would use the API functions `dlopen`, `dlsym`, and `dlclose`. `dlopen` returns a reference to the shared object, `dlsym` finds the location of the function that you want to call, and `dlclose` releases the shared object.

What is returned by the `dlsym` function is actually the memory location of the function in the shared library. To call it, you will need a variable of the appropriate function type.

You will need to declare a function type in order to access the function. Up until now we have been declaring the functions and the shared objects that the functions link to. To demonstrate the powerful nature of plug-ins, Listing 8.8 shows a Kylix implementation of the same plug-in, which uses the same functions the C library does but instead is written in Kylix.

Listing 8.8 - A Kylix implementation of the tax plug-in library

```
library KylixPluginTaxLibrary;

{
    This library is designed to be compatible with the plug-in system
    that was developed to demonstrate dynamically loading shared objects.
    This also demonstrates that the plug-ins can be written in two
    completely different programming languages.

    Written for the "Tomes of Kylix - The Linux API"
    published by Wordware, 2001
}
```

```

uses
    SysUtils;

function GetTaxAmount(Amount: double): double; cdecl; export;
begin
    //Let's return 6.25 percent
    Result := Amount * 0.0625;
end;

procedure GetVersion(var MajorVersion, MinorVersion: integer); cdecl; export;
begin
    MajorVersion := 0;
    MinorVersion := 9;
end;

procedure GetTaxName(TaxName: PChar); cdecl; export;
begin
    StrPCopy(TaxName, 'A Cool Kylix plugin');
end;

exports
    GetTaxAmount,
    GetVersion,
    GetTaxName;

begin
end.

```

Now that we have a C shared object and a Kylix shared object, we should be able to implement the plug-in system. First, however, you will need the procedure types of the functions so that you can use variables of the procedure types to call the functions. The procedure types for the tax system would look like the code in Listing 8.9.

Listing 8.9 - Procedure types used for the plug-in system

```

unit TaxFunctionDeclarations;

interface

type
    TGetTaxAmount = function(Amount: double): double; cdecl;
    TGetVersion = procedure(var MajorVersion, MinorVersion: integer); cdecl;
    TGetTaxName = procedure(TaxName: PChar); cdecl;

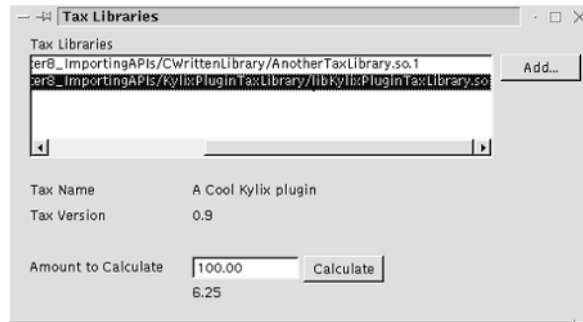
implementation

end.

```

Now that the procedural types have been created, you can use the `dlopen` and `dlsym` functions to return the location of the function calls. The process is relatively straightforward. You first open the library with `dlopen`, then request the location of the tax functions with the `dlsym` function. Once your work is done, close the library with the `dlclose` function. This is what the plug-in system does in Listing 8.10, which demonstrates the full power of creating an application plug-in framework that can be developed using multiple languages.

Figure 8.3 -
The tax
library plug-in
system



Listing 8.10 - Code to dynamically load a shared object

```
unit unitMainPluginForm;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls, Libc;

type
  TfrmTaxPluginSystem = class(TForm)
    lbTaxLibraries: TListBox;
    Label1: TLabel;
    btnAdd: TButton;
    Label2: TLabel;
    lblTaxName: TLabel;
    Label4: TLabel;
    lblTaxVersion: TLabel;
    Label6: TLabel;
    edAmount: TEdit;
    btnCalculate: TButton;
    lblResultAmount: TLabel;
    odTaxLibraries: TOpenDialog;
    procedure btnAddClick(Sender: TObject);
    procedure lbTaxLibrariesClick(Sender: TObject);
    procedure btnCalculateClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmTaxPluginSystem: TfrmTaxPluginSystem;

implementation

uses TaxFunctionDeclarations;

{$R *.xfm}

procedure TfrmTaxPluginSystem.btnAddClick(Sender: TObject);
```

```

begin
    if odTaxLibraries.Execute then
        lbTaxLibraries.Items.Add(odTaxLibraries.FileName);
    end;

procedure TfrmTaxPluginSystem.lbTaxLibrariesClick(Sender: TObject);
var
    LibraryName: array[0..500] of char;
    LibHandle: Pointer;
    GetVersion: TGetVersion;
    GetTaxName: TGetTaxName;
    ErrorMessage1, ErrorMessage2: string;
    TheTaxName: array[0..100] of char;
    majVersion, minVersion: integer;
begin
    if lbTaxLibraries.ItemIndex <> -1 then
        begin
            //Load the details from the tax library.
            StrPCopy(LibraryName, lbTaxLibraries.Items[lbTaxLibraries.ItemIndex]);
            LibHandle := dlopen(LibraryName, RTLD_NOW);
            try
                if LibHandle = nil then
                    showmessage('The Dynamic Library could not be opened.')
                else begin
                    //Obtain the location of the functions within the library
                    GetVersion := dlsym(LibHandle, 'GetVersion');
                    ErrorMessage1 := dlerror;
                    GetTaxName := dlsym(LibHandle, 'GetTaxName');
                    ErrorMessage2 := dlerror;

                    if (ErrorMessage1 <> '') or (ErrorMessage2 <> '') then
                        begin
                            //The function could not be found or there
                            //was an error in the process
                            showmessage('One or both of the functions could not be located.');
```

Exit;

```

                        end;

                        //Write the tax name
                        GetTaxName(TheTaxName);
                        lbTaxName.Caption := TheTaxName;

                        //Write the version
                        GetVersion(majVersion, minVersion);
                        lbTaxVersion.Caption := Format('%d.%d', [majVersion, minVersion]);
                    end;
                finally
                    //Close the library.
                    if LibHandle <> nil then
                        dlclose(LibHandle);
                    end;
                end;
            end;
        end;
    end;

procedure TfrmTaxPluginSystem.btnCalculateClick(Sender: TObject);
var
    LibraryName: array[0..500] of char;
```

```

LibHandle: Pointer;
GetTaxAmount: TGetTaxAmount;
ErrorMessage: PChar;
begin
  if lbTaxLibraries.ItemIndex <> -1 then
    begin
      //Load the details from the tax library.
      StrPCopy(LibraryName, lbTaxLibraries.Items[lbTaxLibraries.ItemIndex]);
      LibHandle := dlopen(LibraryName, RTLD_NOW);
      try
        if LibHandle = nil then
          showmessage('The Dynamic Library could not be opened.')
        else begin
          //Obtain the location of the functions within the library
          GetTaxAmount := dlsym(LibHandle, 'GetTaxAmount');

          //Check for any errors when looking for
          //the functions
          ErrorMessage := dlerror;

          if ErrorMessage <> nil then
            begin
              //The function could not be found or there
              //was an error in the process
              showmessage('The GetTaxAmount function could not be located.');
```

Exit;

```

            end;

            //Display the Calculation
            lblResultAmount.Caption := FloatToStr(
              GetTaxAmount(StrToFloat(edAmount.Text)));
          end;
        finally
          //Close the library.
          if LibHandle <> nil then
            dlclose(LibHandle);
          end;
        end;
      end;
    end;
  end;
End.

```

Shared Object API Functions

dladdr *LibC.pas*

Syntax

```
function dladdr(  
  Handle: Pointer;  
  var Info: TDLInfo;  
):Integer;
```

Description

The `dladdr` function returns information about a shared object handle. The information regarding the shared object is returned in the `TDLInfo` structure.

Parameters

Handle: Handle is the reference value of the shared object that was opened using the `dlopen` function.

Info: Info is a variable of `TDLInfo`.

Return Value

The function returns the location of the function within the shared library. Although the resulting data type is a pointer, it should be typecast to a procedure or function data type that matches exactly the parameters and calling convention of the exported function.

See Also

`dlopen`, `dlclose`, `dlerror`, `dlsym`

dlclose *LibC.pas*

Syntax

```
function dlclose(  
  Handle: Pointer  
):Integer;
```

Description

This function closes a previously opened shared object that was opened with the `dlopen` function. The function decreases the reference count to the shared library, and when no applications are referencing the shared object, the shared object is removed from memory.

Optionally, when the shared object is removed from memory, if there is a function called `_fini`, it is called. Typically, this function is used as a method of freeing any resources that it may have allocated during the process of using the shared object.

Parameters

Handle: Handle is the reference value that was returned from when the library was opened using the `dlopen` function. Make sure that the object itself actually references a shared object. If it doesn't, the error is not handled correctly and may result in a core dump.

Return Value

The function returns the reference count of the shared object after it has been released.

See Also

`dlopen`, `dlsym`, `dlerror`, `dladdr`

Example

See Listing 8.10 - Code to dynamically load a shared object.

dlerror LibC.pas

Syntax

```
function dlerror:PChar;
```

Description

This function is used to test if any problem occurred while using any of the dynamic shared library functions. The message returned is useful for logging messages and determining the problems that occur. Once this message is called, the message is cleared. As a result, calling this method twice in a row will result in clearing the error message.

Return Value

If an error has occurred while using any of the shared object calls, such as `dlopen`, `dlclose`, or `dlsym`, the return value is a pointer to a string that contains a detailed message about what occurred. An example that may be returned as a result of misspelling the function name when using the `dlsym` function would be:

```
/lib/SimpleMathCalls.so.1.0.0: undefined symbol: SimpleAction
```

If, on the other hand, no error occurred, the function simply returns `nil`.

See Also

`dlopen`, `dlclose`, `dlsym`, `dladdr`

Example

See Listing 8.10 - Code to dynamically load a shared object.

dlopen LibC.pas***Syntax***

```
function dlopen(  
  Filename: PChar;  
  Flag: Integer  
): Pointer;
```

Description

This function opens a shared object, which holds a collection of functions and returns a reference to the library. This function must be called before any functions or procedures can be used from the shared object.

If the shared object that is being opened has a routine called “_init” and is not currently loaded into memory, this function will be called. On Delphi-written shared objects, the _init function is exported automatically as the code between the begin and end in the library source code.

Parameters

Filename: Filename is the filename of the shared object or the filename of a linked file to a shared object. Filename can either be the absolute path to the shared object file or it can be the filename only. When it contains only the filename, the Linux API will look for the shared object in the /lib, the /usr/lib, and the cached files located in /etc/ld.so.cache. Passing nil for this parameter makes the library search in its own process.

Flag: Flag refers to how the library functions should be loaded. If the constant RTLD_NOW is used, functions in the shared object are loaded into memory immediately. If the constant RTLD_LAZY is used, the library functions will be opened the first time a function in that library is called.

Return Value

If the shared object cannot be loaded, the function returns nil and information about the error can be found by calling the dlerror function. If the function was successful, a handle to the shared library is returned.

See Also

dlclose, dlsym, dlerror, dladdr

Example

See Listing 8.10 - Code to dynamically load a shared object.

dlsym* LibC.pas**Syntax***

```
function dlsym(  
  Handle: Pointer;  
  Symbol: PChar;  
):Pointer;
```

Description

The `dlsym` function asks for the handle of a particular function in the shared object, so that it can be called within the current application. If the call to `dlsym` is unsuccessful, the details of the error can be found by calling the `dlerror` function.

Parameters

Handle: Handle is the reference value that was returned from when the library was opened using the `dlopen` function.

Symbol: Symbol is the exported function name from the shared object.

Return Value

The function returns the location of the function within the shared library. Although the resulting data type is a pointer, it should be typecast to a procedure or function data type that matches exactly the parameters and calling convention of the exported function.

See Also

`dlopen`, `dlclose`, `dlerror`, `dladdr`

Example

See Listing 8.10 - Code to dynamically load a shared object.

Conclusion

New APIs are fun to play with. There are always times when you have to interface with another vendor's system or maybe there is some cool Linux feature that you want to play with, such as the joystick or the sequencer within Linux. The header files for features such as the joystick or sequencer are already on the system; it is just a matter of doing the conversion to Kylix.

Now that you know the ins and outs of working with shared objects under Kylix, why not create a new Kylix implementation of the library or use your knowledge to extend your application by making a cross-language plug-in system?

References for Kylix Development

This book gives you the knowledge to work comfortably with the Linux API, but there are many other aspects to explore about Linux and Kylix. Included here are many helpful books and Web sites that can assist you in learning more.

Be sure to check <http://www.glennstephens.com.au/tomesofkylix/>, the official home page for this book, for updates to the book as well as bonus material, Kylix classes, and add-ins to assist in your Linux development.

Web Sites

Kylix

The official Kylix home page — <http://www.borland.com/kylix/>

Dr Bob Kylix Kicks (News and info about Kylix) — <http://www.drbob42.com/kylix/index.htm>

The Delphi Jedi home page (Object Pascal API conversions for Windows and Linux) — <http://www.delphi-jedi.org>

The Delphi4Linux home page — <http://www.delphi4linux.org/>

Kylix Component Vendors

Indy Components — <http://www.nevrona.com/indy/>

TurboPower — <http://www.turbopower.com>

Linux Development

GNU (GNU's not UNIX) — <http://www.gnu.org>

The Linux Documentation Project — <http://www.linuxdoc.org>

X Windows Consortium — <http://www.x.org>

Gnome (makers of the Gnome Desktop) — <http://www.gnome.org>

TrollTech Developer Information (makers of the Qt library) — <http://www.trolltech.com/developer/>

Red Hat Package Manager (the most used deployment method on Linux) — <http://www.rpm.org>

Embedded Linux

Linux Devices home page — <http://www.linuxdevices.com/>

Compulab (makers of embedded 80x86 processors) — <http://www.compulab.co.il/>

Red Hat Embedded Development Center — <http://www.redhat.com/embedded/>

Popular Linux Distributions

Red Hat — <http://www.redhat.com>

Caldera Systems — <http://www.calderasystems.com>

Turbo Linux — <http://www.turbolinux.com>

Free BSD — <http://www.freebsd.org>

Slackware Linux — <http://www.slackware.com>

SuSE — <http://www.suse.com>

Debian Linux — <http://www.debian.org/>

Mandrake Linux — <http://www.linux-mandrake.com/en/>

Linux Web Sites

Apache home page (the world's most popular Web server) — <http://www.apache.org>

Open Source (the Open Source Initiative) — <http://www.opensource.org>

Linux.com — <http://www.linux.com>

Linux Online — <http://www.linux.org>

Linux News and Magazines

Linux Magazine — <http://www.linux-mag.com>

Linux Journal — <http://www.linuxjournal.com>

Linux Gazette — <http://www.linuxgazette.com>

Slash Dot — News for nerds <http://www.slashdot.org>

Delphi/Kylix Component Resources

Torry's Delphi Pages — <http://www.torry.ru>

Delphi Super Page — <http://delphi.icm.edu.pl/>

Kylix Compatible Databases

Interbase Certified Edition — <http://www.interbase.com>

Interbase Open Source Edition (Firebird) — <http://www.interbase2000.com>

MySQL (Open Source Database) — <http://www.mysql.com>

Oracle (a scalable database for corporations) — <http://www.oracle.com>

IBM DB2 Universal database for Linux — <http://www-4.ibm.com/software/data/db2/linux/>

Books

Linux Software Development

Bar, Moshe. *Linux Internals*. McGraw-Hill, 2000.

Butenhof, David R. *Programming with POSIX Threads*. Addison Wesley, 1997.

Gallmeister, Bill O. *POSIX.4: Programming for the Real World*. O'Reilly, 1995.

Gay, Warren W. *Advanced UNIX Programming*. Sams Publishing, 2000.

Loosemore, Sandra with Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual Volume One*. Free Software Foundation, 1999.

Robbins, Kay A., and Steven Robbins. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading*. Prentice Hall, 1996.

Stevens, W. Richard. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1993.

Stones, Richard, and Neil Matthew. *Beginning Linux Programming 2nd Edition*. Wrox Press, 1999.

Linux Configuration

Kabir, Mohammed J. *Red Hat Linux 6 Server*. M&T Books, 1999.

Laurie, Ben, and Peter Laurie. *Apache — The Definitive Guide*. O'Reilly, 1997.

Index

- `__chdir`, 51
- `__close`, 29, 51-52
- `__errno_location`, 9, 11
- `__mkdir`, 19, 40, 52-53
- `__pthread_initialize`, 354
- `__raise`, 255
- `__read`, 31, 53-54
- `__rename`, 54
- `__secure_env`, 137, 139
- `__secure_getenv`, 160-161
- `__sigpause`, 287
- `__system`, 46-47, *see also* `system`
- `__time_t`, 44
- `__truncate`, 55
- `__write`, 31-33, 55-56
- `_pthread_cleanup_pop`, 354-355
- `_pthread_cleanup_pop_restore`, 355
- `_pthread_cleanup_push`, 355-356
- `_pthread_cleanup_push_defer`, 356-357

A

- `AbandonSignalHandler`, 224-225
- `abort`, 67
- `accept`, 446, 456
- `alarm`, 255-256
- `alphasort`, 56-57
- `alphasort64`, 56-57
- `append`, 35

B

- `barriers`, 344-348
- `bind`, 445-446, 456-458
- `blocked signals`, 222
- `BlockRead`, 31
- `BlockWrite`, 31
- `buffering`, 46

C

- `CGI`, 33
- `chmod`, 38, 40, 57-58
- `chown`, 38
- `clearenv`, 137, 139, 161
- `clearerr`, 58-59
- `clearerr_unlocked`, 58-59
- `clone`, 357-358
- `closedir`, 45, 59-60
- `closelog`, 159
- `CLX`, 6
 - integrating sockets with, 455
- `condition variables`, 333-335
 - broadcasting, 334
 - creating, 333-334
 - signaling, 334
 - waiting, 334
- `connect`, 436, 458
- `console applications`, 2, 33
- `converting file descriptors and streams`, 34-38
- `creat`, 40, 60-61
- `creat64`, 60-61

D

- `daemon`, 161-162
- `daemons`, 4, 157-158, 160
 - example, 205-210
- `dBase file`, 122
 - file reader, 122-132
- `device drivers`, 4
- `directory`, 45
- `directory streams`, 45
- `dirfd`, 61-62
- `dladdr`, 517
- `dlclose`, 517-518
- `dlopen`, 519
- `dlopen`, 519

dlsym, 520

E

endgrent, 162
 endhostent, 459
 endpwent, 162-163
 endservent, 459
 enumerating users and groups, 153-155
 environment variables, 136-139
 EOSErr, 10
 errno, 9-11
 error codes, 14-18
 errors, 10, 50
 exceptions, 18-25
 exec functions, 150-153
 execl, 151, 163
 execl, 151, 163-165
 execlp, 151, 165-166
 executing Linux commands, 149-150
 execv, 151, 166-168
 execve, 151, 168-170
 execvp, 151, 170-171

F

fchmod, 62-63
 fclean, 63
 fclose, 30, 64, 82
 fcloseall, 64-65
 fcntl, 65-67
 FD_CLR, 485
 FD_ISSET, 485
 FD_SET, 485
 FD_ZERO, 485
 fdopen, 34, 67-69
 feof, 69-70
 feof_unlocked, 69-70
 ferror, 70
 ferror_unlocked, 70
 fexecve, 171-172
 fflush, 46, 70-71
 fflush_unlocked, 70-71
 fgetc, 31, 71-72
 fgetc_unlocked, 71-72
 fgetpos, 72-73
 fgetpos64, 72-73
 fgets, 31-32, 73-74

fgets_unlocked, 73-74
 FIFO, 43
 File, 31
 file conventions, 48-49
 file descriptors, 28-29
 closing, 29
 converting, 34-38
 opening, 29
 reading from, 30-31
 writing to, 31-32
 file errors, 50
 file information, 42-44
 file operations, 46-47
 file ownership, 38-41
 file permissions, 38-41
 displaying, 42-43
 file streams, 28-29
 converting to, 34-38
 fileno, 34, 75-76
 fileno_unlocked, 75-76
 files, 27
 in Linux, 28
 opening, 29-30
 reading/writing, 30-32
 fopen, 28, 30, 76-77, 82
 fopen64, 76-77
 fork, 172-173, 448-449
 forking, 140-143
 fprintf, 77-78
 fputc, 31, 78-79
 fputc_unlocked, 78-79
 fputs, 31, 79-80
 fputs_unlocked, 79-80
 fread, 31-32, 80-82
 fread_unlocked, 80-82
 freopen, 82-83
 freopen64, 82-83
 fseek, 83-84
 fseeko, 83-84
 fseeko64, 83-84
 fsetpos, 84-85
 fsetpos64, 84-85
 fstat, 42, 85-86
 fstat64, 42, 85-86
 ftell, 86-87
 ftello, 86-87

ftello64, 86-87
 ftok, 254
 fwrite, 31-32, 87-89
 fwrite_unlocked, 87-89

G

get_current_dir_name, 89
 getc, 31, 89-90
 getc_unlocked, 89-90
 GetCurrentThreadID, 358
 getcwd, 90-91
 getdelim, 32, 91-93
 getegid, 173
 getenv, 137, 139, 173-174
 geteuid, 174-175
 getgid, 175
 getgrent, 155, 175-176
 getgrgid, 177
 getgrnam, 178-179
 getgroups, 179
 gethostbyaddr, 459-461
 gethostbyname, 461-463
 gethostbyname2, 463-464
 gethostent, 464-465
 gethostname, 465-466
 GetLastError, 18
 getline, 31, 93-94
 getpeername, 466-467
 getpgid, 180
 getpgrp, 180-181
 getpid, 134, 181
 getppid, 134, 181-182
 getprotobyname, 467-468
 getprotobynumber, 468-469
 getpwent, 154, 182-183
 getpwnam, 183-184
 getpwuid, 134-135, 184-185
 getservbyname, 469-470
 getservbyport, 470-472
 getservent, 472-473
 getsockname, 473-474
 getsockopt, 475-477
 getuid, 185-186
 getw, 94-95
 getwd, 95-96
 GLibC, 5

GNOME, 3-4
 GNU, 5
 group_member, 186-187

H

handles, transferring, 35-36
 HookSignal, 223-225
 htonl, 435, 445, 477-478
 htons, 435, 436, 444, 478-479

I

importing C-written functions, 509-511
 inet_addr, 436, 444, 479
 inet_ntoa, 480
 inetd, 454
 InquireSignal, 224-225
 interprocess communication, 211-212
 different methods of, 212-214
 ioctl, 47-48, 96-97
 IPC keys, 253-254

K

KDE, 3
 kernel, 10
 kill, 147, 217-218, 256-257
 killpg, 257-258
 Kyxix, 6-7
 resources, 521-523

L

LibC, 5
 linked files, 48
 Linux, 1
 API, 5
 applications, 2-4
 errors, 9, 18, 50
 files in, 28
 history, 1-2
 kernel, 4
 shortcuts, 48
 vs. Windows, 7
 Linux commands, executing, 149-150
 listen, 446, 480-481
 ln, 48
 lseek, 97-98
 lseek64, 97-98
 lstat, 42, 98-100

lstat64, 42, 98-100

M

message queues, 214, 233-239
 example, 295-302

MkDir, 40

mkfifo, 229-230, 258

msgctl, 234-235, 259-260

msgget, 234, 260-261

msgrcv, 234-235, 261-262

msgsnd, 234-235, 262-263

mutexes, 322-326

N

named pipes, 214, 229-233

ntohl, 435, 481

ntohs, 435, 481-482

O

open, 29-30, 49, 100-102

open_memstream, 102-103

open64, 49, 100-102

opendir, 45, 103

openlog, 158

ownership, 38-41

P

pause, 263

pclose, 263-266

PDiretoryStream, 45, 60

PDirent, 45

PDireInfo, 60

permissions, 38-41

 displaying, 42-43

perror, 11-13, 446

PIOFile, 30

pipe, 225, 264

pipes, 213-214

 programming, 225-229

popen, 228-229, 265-266

POSIX, 5

POSIX threads, 305-306

 synchronizing with, 319-326

pread, 103-104

printf, 49

process groups, 143

process ID, 133

processes, 133

 and environment variables, 136-139

 communicating between, 211-212

 forking, 140-143

 gathering information about, 134-135

 terminating, 147-149

 vs. threads, 308-309

 waiting, 144-147

psignal, 266-267

pthread_atfork, 358-360

pthread_attr_destroy, 312, 360-361

pthread_attr_getdetachstate, 312, 361

pthread_attr_getguardsize, 312, 362

pthread_attr_getinheritsched, 312, 362-363

pthread_attr_getschedparam, 312, 363-364

pthread_attr_getschedpolicy, 312, 364

pthread_attr_getscope, 312, 365

pthread_attr_getstack, 312, 365-366

pthread_attr_getstackaddr, 312, 366-367

pthread_attr_getstacksize, 312, 367

pthread_attr_init, 312-313, 367-368

pthread_attr_setdetachstate, 312-313, 368

pthread_attr_setguardsize, 312, 368-369

pthread_attr_setinheritsched, 312, 314, 369-370

pthread_attr_setschedparam, 312, 314, 370-371

pthread_attr_setschedpolicy, 312, 314, 371-372

pthread_attr_setscope, 312, 314, 372

pthread_attr_setstack, 312, 372-373

pthread_attr_setstackaddr, 312, 373-374

pthread_attr_setstacksize, 312, 374

pthread_barrier_destroy, 374-375

pthread_barrier_init, 375-376

pthread_barrier_wait, 376

pthread_barrierattr_destroy, 376-377

pthread_barrierattr_getpshared, 377

pthread_barrierattr_init, 377-378

pthread_barrierattr_setpshared, 378

pthread_cancel, 318, 378-380

pthread_cond_broadcast, 333-334, 380

pthread_cond_destroy, 333, 380-381

pthread_cond_init, 333-334, 381

pthread_cond_signal, 333-334, 382

pthread_cond_timedwait, 333, 382-383

pthread_cond_wait, 333-334, 383-384

pthread_condattr_destroy, 384

pthread_condattr_getpshared, 384-385

pthread_condattr_init, 385
 pthread_condattr_setpshared, 385-386
 pthread_create, 310-311, 313, 317, 386-387
 pthread_detach, 387-389
 pthread_equal, 389
 pthread_exit, 390-391
 pthread_getschedparam, 391-392
 pthread_getspecific, 348-349, 392
 pthread_join, 317, 393
 pthread_key_create, 348, 393-394
 pthread_key_delete, 349, 394-395
 pthread_kill, 395
 pthread_kill_other_threads_np, 352, 396
 pthread_mutex_destroy, 396
 pthread_mutex_init, 325, 396-397
 pthread_mutex_lock, 325, 397-398
 pthread_mutex_timedlock, 398
 pthread_mutex_trylock, 399
 pthread_mutex_unlock, 399-400
 pthread_mutexattr_destroy, 400
 pthread_mutexattr_getpshared, 400-401
 pthread_mutexattr_gettype, 401-402
 pthread_mutexattr_init, 402
 pthread_mutexattr_setpshared, 402-403
 pthread_mutexattr_settype, 403-404
 pthread_once, 335-336, 404
 pthread_rwlock_destroy, 404-405
 pthread_rwlock_init, 405-406
 pthread_rwlock_rdlock, 406
 pthread_rwlock_timedrdlock, 407
 pthread_rwlock_timedwrlock, 407-408
 pthread_rwlock_tryrdlock, 408-409
 pthread_rwlock_trywrlock, 409
 pthread_rwlock_unlock, 409-410
 pthread_rwlock_wrlock, 410
 pthread_rwlockattr_destroy, 410-411
 pthread_rwlockattr_getpshared, 411
 pthread_rwlockattr_init, 412
 pthread_rwlockattr_setpshared, 412-413
 pthread_self, 413
 pthread_setcancelstate, 318, 413-416
 pthread_setcanceltype, 318, 417
 pthread_setschedparam, 417-418
 pthread_setspecific, 348-349, 419
 pthread_sigmask, 419-420
 pthread_spin_destroy, 420-421

pthread_spin_init, 421
 pthread_spin_lock, 421-422
 pthread_spin_trylock, 422
 pthread_spin_unlock, 422-423
 pthread_testcancel, 318, 423
 pthread_yield, 423-424
 PUnixTime, 44
 putc, 31, 78-79
 putenv, 137, 139, 187-188
 putw, 104-105
 pwrite, 105

Q

Qt, 3-4

R

RaiseLastError, 10
 read, 31
 read/write locks, 336-337
 readdir, 45, 105-107, 309
 readdir_r, 309
 readdir64, 105-107
 readln, 31
 recv, 482-483
 recvfrom, 483-484
 reentrant functions, 309
 remove, 107
 reset, 35
 rewind, 107-108
 rewinddir, 108-109

S

scandir, 109-111
 scandir64, 109-111
 seek, 46
 seekdir, 46, 111-112
 select, 484-489
 sem_close, 326, 424
 sem_destroy, 326, 424-425
 sem_getvalue, 326, 425
 sem_init, 326, 425-426
 sem_open, 326, 426-427
 sem_post, 326, 327, 427
 sem_timedwait, 326, 428
 sem_trywait, 326, 428-429
 sem_unlink, 326, 429
 sem_wait, 326-327, 429-430

- semaphores, 240, 324-327
 - programming, 240-245
 - using with shared memory, 248-253
- semctl, 240-241, 267-271
- semget, 240-241, 271
- semop, 240-241, 271-273
- send, 491-492
- sendto, 492-495
- setbuf, 112-113
- setbuffer, 113-114
- setegid, 188
- setenv, 137, 139, 189-190
- seteuid, 190-191
- setgid, 191-192
- setgrent, 192
- sethostent, 490
- setlinebuf, 114-115
- setpgid, 192-193
- setpgrp, 193
- setpwent, 194
- setregid, 194-195
- setreuid, 195-196
- setservent, 490-491
- setsockopt, 495-496
- setuid, 196-197
- setvbuf, 46, 115-116
- shared memory, 214, 245-246
 - programming, 246-248
 - using with semaphores, 248-253
- shared objects, 503
 - creating with C, 508-511
 - creating with Kylix, 504-505
 - dynamically loading, 512-516
 - using, 506-508, 511-512
- shmat, 246-247, 273-274
- shmctl, 247-248, 274-275
- shmdt, 247, 276
- shmget, 246-247, 276-277
- shutdown, 496-497
- sigaction, 219, 277-278
- sigaddset, 221, 278
- sigandset, 221, 279-280
- sigdelset, 221, 280
- sigemptyset, 221, 280-281
- sigfillset, 221, 281-282
- siggetmask, 282
- sighold, 282-283
- sigignore, 283
- sigisemptyset, 221, 283-284
- sigismember, 221, 284-285
- signal, 285-286
- signal mask, 223
- signal sets, 218-221
- signals, 10, 212, 215-218
 - and threads, 351-352
 - blocking, 221-223
 - sending, 217-218
- sigorset, 221, 286-287
- sigpending, 223, 288
- sigprocmask, 222, 288-290
- sigrelse, 290-291
- sigset, 291
- sigsuspend, 292
- sigtimedwait, 294-295
- sigwait, 293
- socket, 433-434, 497-498
- sockets, 431-432
 - accepting, 446-448
 - addresses, 434
 - client, 436-438
 - communicating with, 439-442
 - creating, 433
 - creating server, 442-445
 - establishing connection, 432-433
 - integrating with CLX, 455
 - listening to, 446-448
 - naming, 445-446
 - processing, 448-453
- socketpair, 498-501
- spin locks, 341-344
- standard error, 33
- standard input, 33
- standard output, 33
- stat, 42, 116-117
- stat64, 42, 116-117
- streams, 28-30
 - closing, 30
 - converting to, 34-38
 - opening, 30
 - reading from, 32

- writing to, 32
- strerror, 9, 13
- strsignal, 294
- superservers, 453
- synchronization methods, 319-326, 336-337
- SysErrorCode, 10
- syslog, 158
- system, 149, 197, *see also* __system

T

- tell, 46
- telldir, 46, 117-118
- tempnam, 118-119
- TextFile, 31
- TFileRec, 35
- TFileStream, 38
- THandleStream, 36-38
- thread-specific data, 348-349
- threads, 305-306
 - and signals, 351-352
 - attributes, 311-317
 - choosing, 308-309
 - creating, 310-311
 - errors, 309
 - integrating with TThread, 352-354
 - killing, 318-319
 - Linux Threads, 306
 - POSIX threads, 305-306
 - synchronization, 319-326, 336-337
 - vs. processes, 308-309
 - waiting on, 317
- time, converting, 44
- TIOFile, 29
- tmpfile, 119
- tmpfile64, 119
- tmpnam, 120
- tmpnam_r, 120
- Torvalds, Linus, 1
- Trolltech, 3
- TSemaphoreBuffer, 241-242
- TSigSet, 219

- TStatBuf, 42, 45
- TStream, 29, 36
- TUnixTime, 44

U

- umask, 357
- ungetc, 120-121
- UnHookSignal, 224-225
- UNIX, 5
- unlocked functions, 49
- unsetenv, 198
- user ID, 134

V

- varargs, 47, 49-50
- VCL, 6
- versionsort, 122
- versionsort64, 122
- vfork, 198-199
- Visual Component Library, *see* VCL

W

- wait, 145, 199-200
- wait3, 145, 201-202
- wait4, 145, 203-204
- waitpid, 145, 204-205
- WCOREDUMP, 145-146
- WEXITSTATUS, 145
- WIFEXITED, 145
- WIFSIGNALED, 145-146
- WIFSTOPPED, 145-146
- Windows vs. Linux, 7
- write, 31, 35
- WSTOPSIG, 145-146
- WTERMSIG, 145

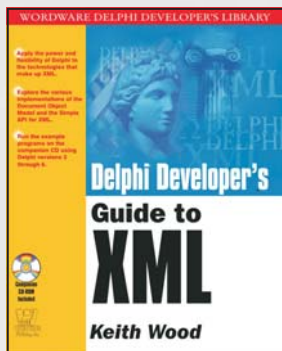
X

- X/Open, 5
- xinetd, 455
- XWindows, 2-4, 33

Looking for more?

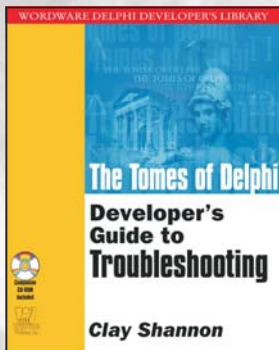
Check out Wordware's market-leading Delphi Developer's Library featuring the following new releases and upcoming titles.

Available Now:



Delphi Developer's Guide to XML

1-55622-812-0
\$59.95 • 7½ x 9¼
544 pp.



The Tomes of Delphi: Developer's Guide to Troubleshooting

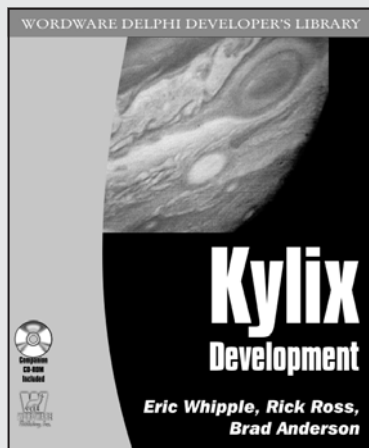
1-55622-816-3
\$59.95 • 7½ x 9¼
568 pp.



The Tomes of Delphi: Win32 Core API

1-55622-750-7
\$59.95 • 7½ x 9¼
760 pp.

Coming Soon:



Kylix Development

1-55622-774-4
\$49.95
7½ x 9¼
600 pp.

Available January 2002

Check out the complete Delphi Library online at
www.wordware.com

About the CD

The companion CD contains source code for the examples, as well as source code for the Linux API, the Kylix Open Edition, and the Kylix 2 trial edition. These are organized in the following directories:

- BookSource Code—source code for the example applications in the book.
To extract the source files from the archive, enter:
`gunzip TomesOfKylixSource.tar.gz`
`tar -xf TomesOfKylixSource.tar`
- GLibCSrc—source code to the Linux API.
- KylixOpenEdition—the Open Edition of Kylix. Follow the instructions in that directory to install this product.
- KylixTrialEdition—the Kylix 2 trial edition. Follow the instructions in that directory to install this product.

For more information, see `readme.txt` on the CD.

If you have any questions or comments about this book, please e-mail the author at tomesofkylix@glennstephens.com.au or visit <http://www.glennstephens.com.au/tomesofkylix/>.

Warning:

By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement found on the following page.

Additionally, opening the CD package makes this book nonreturnable.

CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.

Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement.

Additionally, opening the CD package makes this book non-returnable.